

Data Consistency and Coordination for Untrusted Environments

**Vom Fachbereich Informatik der Technischen Universität Darmstadt
genehmigte Dissertation zur Erlangung des akademischen Grades eines**

Doktor rerum naturalium (Dr. rer. nat.)

vorgelegt von

Dipl.-Inform. Matthias Majuntke

geboren in Dernbach (Westerwald)

Referenten:

Prof. Neeraj Suri, PhD

Prof. Christof Fetzer, PhD

Datum der Einreichung: 18.07.2012

Datum der mündlichen Prüfung: 21.09.2012

Hochschulkennziffer: D17

Darmstadt 2012

Abstract

Users of today's computing devices are accustomed to having a permanent and capable connection to the Internet. Personal data and computational tasks are increasingly assigned to online services. Besides many advantages, online services may not be fully trusted by the users as they are usually hosted by a third party provider. Cryptographic techniques are able to prevent a provider from leaking or modifying sensitive user data. However, other attacks are still possible: When clients interact only through an untrusted online service, the latter may send diverging and inconsistent replies. In this context, fork-consistent semantics make it much easier for the clients to detect such violations. They ensure that if an untrusted service only *once* sent a wrong response to some client, then this client remains *forever forked* from those other clients to which the service replied differently. If fork-consistency is provided, clients may easily detect service misbehavior by out-of-band communication.

Recent research results have shown that it is impossible to implement a service that provides full consistency and *wait-free* operations in the fault-free case and gracefully degrades to *fork-linearizability*, the strongest notion of fork-consistency, if the service acts maliciously. All existing solutions are based on locks, and thus, client operations may block even if the service is correct. This thesis introduces the first *lock-free* implementations with *fork-linearizability*, providing *abortable* (and therefore *obstruction-free*) operations if the service behaves correctly. In practical settings, obstruction-free solutions can easily be boosted to *wait-freedom*.

In the context of fork-consistency, the thesis demonstrates that the underlying system assumptions can be significantly reduced. Existing works require the shared service to execute non-trivial computation steps. In the thesis at hand it is shown that for a wide range of fork-consistent implementations a service providing only a simple read/write interface is sufficient. For practical systems this makes a big difference in cost as full-fledged servers are typically more expensive than simple storage devices.

The second part of this thesis deals with the orthogonal question how to implement shared storage abstractions that do not exhibit malicious, i.e., Byzantine faulty, behavior. The basic principle is to achieve Byzantine fault-tolerance by replication over a set of replicas out of which a fraction may act maliciously. The thesis presents *lightweight*, Byzantine fault-tolerant implementations of an atomic register and a key-value-store as required for many modern services in the cloud. The notion of *lightweight* comprises several aspects to reduce the costs incurred by replication, e.g., a minimal number of replicas and communication rounds, no employment of self-verifying data, and the support of an unbounded number of possible malicious readers.

Kurzfassung

Die Nutzer aktueller Computergeräte erwarten einen permanenten und leistungsfähigen Zugang zum Internet. Deshalb werden persönliche Daten und Aufgaben zunehmend von Online-Services verarbeitet. Neben vielen gebotenen Vorteilen können Benutzer solchen Services jedoch nicht vollumfänglich vertrauen, da diese üblicherweise von Drittanbietern bereitgestellt werden. Mithilfe kryptographischer Techniken kann eine nichtauthorisierte Offenlegung oder Änderung von Benutzerdaten durch den Service-Anbieter wirkungsvoll verhindert werden. Andere Angriffe sind jedoch weiterhin möglich: Wenn Clients nur durch einen nicht vertrauenswürdigen Online-Service kommunizieren, kann dieser von einander abweichende und inkonsistente Antworten an die Clients versenden. In diesem Kontext erleichtert die Eigenschaft *Verzweigungskonsistenz* (*fork-consistency*) es den Clients, ein solches Fehlverhalten des Services zu erkennen. Verzweigungskonsistenz garantiert, dass, sobald ein nicht vertrauenswürdiger Service eine inkonsistente Antwort an einen Client sendet, dieser Client für immer *isoliert* (und damit *verzweigt*) von den anderen Clients bleibt. In einem System, das Verzweigungskonsistenz erfüllt, können Clients einen fehlerhaft agierenden Server durch systemexterne Kommunikation leicht detektieren.

Aktuelle Forschungsergebnisse zeigen, dass es nicht möglich ist einen Service zu implementieren, der volle Konsistenz und unabhängig terminierende (*wait-free*) Operationen im fehlerfreien Fall ermöglicht und, falls sich der Service bösartig verhält, niemals *Verzweigungslinearisierbarkeit*, die stärkste verzweigungskonsistente Eigenschaft, verletzt. Alle bislang existierenden Ansätze benötigen dafür *Locks*, so dass Operationen der Clients selbst dann blockieren können, wenn der Service korrekt ist. Diese Dissertation stellt die ersten lock-freien Implementierungen mit *Verzweigungslinearisierbarkeit* vor, die nebenläufige Operationen *abbrechen* um ein Blockieren zu verhindern. Für praktische Anwendungen können solche abbrechbaren Operation leicht in unabhängig terminierende Operationen überführt werden.

Weiterhin zeigt die vorliegende Dissertation wie im Kontext von Verzweigungskonsistenz die zugrundeliegenden Systemannahmen signifikant reduziert werden können. In existierenden Ansätzen muss der benötigte, gemeinsam genutzte Service nichttriviale Berechnungsschritte ausführen. In dieser Arbeit wird gezeigt, dass für ein großes Spektrum von verzweigungskonsistenten Implementierungen ein Service ausreicht, der nur eine einfache Schreib/Lese-Schnittstelle bereitstellt. Für Systeme in der Praxis besteht ein großer Kostenunterschied zwischen vollwertigen Servern und einfachen Speichermodulen.

Der zweite Teil der vorliegenden Dissertation behandelt die orthogonale Fragestellung, wie gemeinsam genutzter Speicher implementiert werden kann, der kein fehlerhaftes Verhalten zeigt. Das Grundprinzip ist, durch Replikation Fehlertoleranz gegenüber beliebigen Fehlertypen zu erreichen. Das bedeutet, dass aus einer Menge von replizierten Servern ein Teil der Server beliebiges, fehlerhaftes Verhalten aufweisen kann. In dieser Arbeit werden effiziente, fehlertolerante Implementierungen von *atomischen Registern* und *Key-Value-Stores* vorgestellt. Key-Value-Stores werden hauptsächlich in vielen modernen Cloud-

Services eingesetzt. Die vorgestellten Implementierungen reduzieren in mehrfacher Hinsicht den durch Replikation verursachten Overhead, z.B. durch eine minimale Anzahl von Replikas und Kommunikationsrunden, den Verzicht auf Datenauthentifizierung sowie die Unterstützung einer beliebigen Anzahl von möglicherweise bösartigen Lese-Clients.

Acknowledgements

First of all, I would like to thank my supervisor Prof. Neeraj Suri. Throughout all the years I was working with Neeraj, he gave me the freedom and responsibility to follow my own interests and to built up my research profile. Thank you for providing me access to the research community and for keeping most of the project work away from me. I thank my second supervisor Prof. Christof Fetzer for taking the time of reading through all the technical proofs of this thesis.

I am deeply grateful to my closest colleagues Dan Dobre and Marco Serafini for the creative and tough discussions (also the non-technical ones) we had in our “distributed systems” subgroup. Marco and Dan are probably the most skeptical persons I ever met which significantly improved my research as all ideas had to first convince Dan and Marco. Especially, I would like to thank Dan, who has become a close friend, for our cycling and skiing trips and for keeping common research projects going even after he left the DEEDS group.

I would like to thank my close colleague Peter Bokor for his many helpful and critical comments on my research work. Although having a different research background he always took the time to understand and consider my problems. Special thanks go to my former and current office mates Andreas, Marco, Dan, Peter, Stefan, and Arda for the great time we had in E221, the only office that actually deserves the “AWESOMENESS” poster hanging on the wall. Thanks to the Mensa-boycotting folks Piotr, Vinay, Jesus, Arda for the fun during lunch time. A big “Thank You” to all DEEDS members for making everyday-work in the group such a pleasant time, to Sabine for doing all the paperwork, and to Ute for keeping the DEEDS “Stammtisch” project running.

Finally, I would like to thank my wife Verena for all her love and support, and her patience with my occasional grumpiness. Nobody else could understand better all the ups and downs I was experiencing when doing my PhD.

Matthias

Mannheim, September 7th, 2012

Preface

This thesis comprises the research results I developed as a PhD student under the supervision of Prof. Neeraj Suri, PhD in the DEEDS group at Computer Science Department, Technische Universität Darmstadt during the time from end of 2006 until Summer 2012. My research covers the area of untrusted system environments that can be partitioned into two main directions that are the focus of this thesis: Firstly, distributed protocols that implement shared functionalities with fork-consistent semantics on top of a maliciously behaving server. Secondly, implementations of shared storage that achieve Byzantine fault-tolerance through replication over a collection of servers out of which a subset may fail. The thesis at hand is a compilation of three published conference papers [MDSS09, MDS11, MDSCS11] and a technical report that will be submitted for publication [MDS12].

[MDSCS11] Matthias Majuntke, Dan Dobre, Christian Cachin, and Neeraj Suri. *Fork-Consistent Constructions From Registers*. In Proceedings of the 15th International Conference on Principles of Distributed Systems (OPODIS), Toulouse, France, 2011.

[MDS11] Matthias Majuntke, Dan Dobre, and Neeraj Suri. *Fork-Consistent Constructions From Registers* (Brief Announcement). In Proceedings of the 30th annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), San Jose, California, USA, 2011.

[MDSS09] Matthias Majuntke, Dan Dobre, Marco Serafini, and Neeraj Suri. *Abortable Fork-Linearizable Storage*. In Proceedings of the 13th International Conference on Principles of Distributed Systems (OPODIS), Nimes, France, 2009.

[MDS12] Matthias Majuntke, Dan Dobre, and Neeraj Suri. *Lightweight Robust Atomic Storage*. Technical Report. Technische Universität Darmstadt. TR-TUD-DEEDS-07-01-2012. July, 2012.

During my time as a PhD student I further worked on lower bounds for fault-tolerant storage implementations [DGM⁺11, DGM⁺12], on Byzantine and crash fault-tolerant state machine replication protocols [DMSS10, SBD⁺10], on eventually linearizable shared objects that allow concurrent access [SDM⁺10], and on storage and latency efficient shared registers [DMS08, DMSS09].

[DGM⁺12] Dan Dobre, Rachid Guerraoui, Matthias Majuntke, Neeraj Suri, Marco Vukolić. *The Complexity of Robust Storage*. In Journal of the ACM (JACM), 2012 (submitted).

[DGM⁺11] Dan Dobre, Rachid Guerraoui, Matthias Majuntke, Neeraj Suri, Marco Vukolić. *The Time-Complexity of Robust Atomic Storage*. In Proceedings of the 30th annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), San Jose, California, USA, 2011.

[SDM⁺10] Marco Serafini, Dan Dobre, Matthias Majuntke, Peter Bokor, Neeraj Suri. *Eventually Linearizable Shared Objects*. In Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), Zurich, Switzerland, 2010.

[SBD⁺10] Marco Serafini, Peter Bokor, Dan Dobre, Matthias Majuntke, Neeraj Suri. *Scrooge: Reducing the Costs of Fast Byzantine Replication in Presence of Unresponsive Replicas*. In Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN), Chicago, Illinois, USA, 2010.

[DMSS10] Dan Dobre, Matthias Majuntke, Marco Serafini, Neeraj Suri. *HP: Hybrid Paxos for WANs*. In Proceedings of the 2010 European Dependable Computing Conference (EDCC), Valencia, Spain, 2010.

[DMSS09] Dan Dobre, Matthias Majuntke, Marco Serafini, Neeraj Suri. *Efficient Robust Storage Using Secret Tokens*. In Proceedings of the 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS), Lyon, France, 2009.

[DMS08] Dan Dobre, Matthias Majuntke, Neeraj Suri. *On the Time-Complexity of Robust and Amnesic Storage*. In Proceedings of the 12th International Conference on Principles of Distributed Systems (OPODIS), Luxor, Egypt, 2008.

Contents

Abstract	iii
Kurzfassung	v
Acknowledgements	vii
Preface	ix
1 Introduction	1
1.1 Thesis in a Nutshell	1
The Times They are A-Changing	1
Caught In The Act	2
Mission Impossible	3
How to Get Around	5
Better is the Enemy of Good	6
Make it Easy	8
There ain't no such thing as a free lunch	9
A Good Deal	10
Going Deeper	12
No One Knows What the Future Holds	13
The Wheel is Come Full Circle	14
1.2 Contributions in a Nutshell	15
1.2.1 Abortable Fork-Linearizable Storage	17
1.2.2 Concurrent Abortable Fork-Linearizable Storage	18
1.2.3 An Abortable Fork-Linearizable Universal Object from Reg- isters	19
1.2.4 Wait-free Weak-Fork-Linearizable Storage from Registers	20
1.2.5 A Lightweight Robust Atomic Register	21
1.2.6 A Lightweight Robust Key-Value Store	22
2 Preliminaries	23
2.1 System Model	23
2.1.1 Basic Concepts	23
2.1.2 Implementation of Shared Objects	24

2.1.3	Sequential Specification of Shared Objects	25
2.1.4	Communication Models	27
2.2	Background	28
2.2.1	Comparing Servers	28
2.2.2	Fork-Consistency	30
3	Abortable Fork-Linerizable Storage	33
3.1	Introduction	34
3.2	Related Work	35
3.3	System Model and Definitions	36
3.4	Preliminaries on the Protocols	39
3.5	(C1): The LINEAR Protocol	40
3.5.1	Description of the LINEAR Protocol	40
3.5.2	Correctness Arguments	44
3.5.3	LINEAR Protocol Proof	44
3.6	(C2): The CONCUR Protocol	51
3.6.1	Description of the CONCUR Protocol	51
3.6.2	Correctness Arguments	53
3.6.3	CONCUR Protocol Proof	54
3.7	Analysis & Conclusion	60
4	Fork-Consistent Emulations from Registers	63
4.1	Introduction	64
4.2	Related Work	66
4.3	System Model	67
4.4	(C3): The AFL Protocol	70
4.4.1	Protocol Ideas	70
4.4.2	Description of the AFL Protocol	72
4.4.3	Correctness Arguments	72
4.4.4	Proof of Correctness of the AFL Protocol	74
4.5	(C4): The WFL Protocol	81
4.5.1	Protocol Ideas	81
4.5.2	Description of the WFL Protocol	83
4.5.3	Correctness Arguments	85
4.5.4	Proof of Correctness of the WFL Protocol	86
4.6	Analysis & Conclusion	92
5	Lightweight Atomic Storage	95
5.1	Introduction	96
5.1.1	Atomic Registers	96
5.1.2	Key-Value Store	98

5.2	Related Work	99
5.3	System Model	101
5.4	(C5): The LwR Protocol	102
5.4.1	Idea of the LwR Protocol	102
5.4.2	Description of the LwR Protocol	107
5.4.3	Proof of Correctness of the LwR Protocol	111
5.5	(C6): The LwKVS Protocol	117
5.5.1	From Atomic Registers to Key-Value Stores	117
5.5.2	Description of the LwKVS Protocol	120
5.5.3	Definition of Regular List Operations	128
5.5.4	Proof of Correctness of the LwKVS Protocol	129
5.6	Conclusion & Discussion	138
6	Conclusion	141
	Abortable Fork-Linearizable Storage	141
	Fork-Consistent Emulations from Registers	143
	Lightweight Atomic Storage	144
	The Final Word	146
	Bibliography	149
	List of Algorithms	161
	List of Figures	163
	Curriculum Vitae	165

Contents

1 Introduction

1.1 Thesis in a Nutshell

The Times They are A-Changing

In the last decade, we have observed the increasing trend that a wide range of computing devices has a nearly permanent broadband access to the Internet. This development has fundamentally changed the way how computers are used nowadays and enabled the origin of a new kind of online services: Instead of storing and processing data only on the local device, users assign these tasks to online services “in the cloud”. This paradigm shift exists for different types of devices ranging from smart phones, tablets, and personal computers to server systems. The range of offered services is equally wide: Simple online services like storage, email, online documents [Goo] and further upcoming Web 2.0 applications [YWG⁺08]; applications for online collaboration like the revision control systems CVS [CVS] and SVN [SVN]; storage management systems like WebDAV [Whi97] and a large number of distributed file systems [Wik]; the full “cloud” services like Amazon S3, Nirvanix CloudNAS, and Microsoft SkyDrive [CKS09b] that provide Software-, Platform-, and even Infrastructure-as-a-Service (SaaS, PaaS, IaaS).

Users benefit from such online services as they offer full data administration. Hence, the user does not need to care for backups, software updates or server maintenance. From an economic point of view, the expenses for setting up and running a whole server infrastructure can be outsourced. Moreover, the data is always available and can be accessed from everywhere. The latter also makes online collaboration, where multiple users work on the same logical data, very attractive. For instance, it enables cooperation among agencies of a large company or it allows members of a project from different countries or time zones to work on the same shared data.

However, all these key benefits of online services are at the same time a major issue — the services are provided by a third party, i.e., they are usually not inside the user’s administrative domain. Thus, the user may not fully trust the online service as the service provider might corrupt or leak sensitive data in many ways: It may grant unauthorized access to confidential data, data might be altered undetectably, different users might have inconsistent views of the service, and the provider might refuse to grant access to user data at all. Such malicious

behaviors can be classified as violations of confidentiality, integrity, consistency, and availability. Cryptographic encryption techniques are able to prevent unauthorized access to data (confidentiality). Digital signatures, hash functions, and message authentication codes (MACs) ensure that each data corruption becomes easily detectable, thus ensuring integrity. Availability, however, cannot be guaranteed in case the service provider is untrusted, as it may simply refuse to reply to user requests. The only possibility to compensate a not responding service is for the user to replicate the data over multiple service providers which may be then used as backups.

A similar problem as with availability exists for the consistency of data: A user cannot prevent that an untrusted service does not return the most recent but stale data to the user. Such an attack especially arises when several users are collaborating on the same logical data: a recent update of one user is omitted and an outdated value is presented to another user. This split brain attack is called *forking* and cannot be prevented. Starting from a forking attack an even worse situation can appear in the presence of an untrusted service. Let Alice and Bob be two users of some untrusted service that allows online collaboration on text documents. Alice stores her most recent version of the shared document at the service, while the service presents an outdated version of Alice's document to Bob. Bob is unable to distinguish an up-to-date version from an outdated one and thus, from his point of view no malicious behavior has appeared. Bob proceeds by editing the shared document and uploading his newest version back to the service. Finally, Bob's version is now presented to Alice. In this situation, Alice finds a document which lacks her newest contributions (that have been omitted by the untrusted service) but where Bob also did his editing. This means, that Alice has to compare Bob's version with her most recent version to find out that some of her updates are missing. However, these changes might also have been done by Bob. Unless Alice does not do such a consistency check with her own versions and as long there is no out-of-band communication with Bob (e.g., per phone or email), the misbehavior of the service cannot be easily detected, making reliable collaboration impossible. In such a situation *fork-consistent* semantics, which is a main topic of the thesis at hand, come into play to preclude that an untrusted service may raise such an attack without being caught in the act.

Caught In The Act

Fork-Consistency is a safety property implemented among the users (i.e., clients) and the untrusted service. Intuitively, it ensures that once two clients are forked, they never see each others' updates after that, or they may reveal the service as faulty. Hence, with respect to consistency it remains for the malicious service only to partition the users' views of the system by omitting updates without being

detected. Once such a partitioning occurs, the users stop hearing from each other. If Alice has not seen updates from user Bob for a while she may now use out-of-band communication (e.g., phone or e-mail) to easily find out whether the server is misbehaving or if Bob did not do any updates.

Fork-consistent semantics comprise a number of properties that can be ordered according to their strength. *Fork-linearizability* [MS02] is the strongest existing fork-consistent property. It is based on the well-known consistency property *linearizability* [HW90]. *Linearizability* is defined as a property on shared objects that may be concurrently accessed by several clients. A shared object provides operations at its API that can be called by the clients. Two operations (of two clients) are said to be *concurrent* if the one is initiated while the other one has not yet finished. Each operation may have some input parameters, changes the state of the shared object, and returns a result to the client. Hence, the operations occur on a shared object as an interleaved sequence of invocation and response events. Such a shared object satisfies *linearizability* if the sequence of operations can be ordered such that operations appear one after the other while neither the real-time ordering nor the sequential specification of the object is violated — e.g., if a read operation returns some value v from the object, the corresponding operation writing value v has to be scheduled before the read operation. As an untrusted service can also be seen as a shared object, *linearizability* would be violated by omitting updates and presenting it later on. Linearizability is a very desirable property as it gives the illusion of atomic operations taking effect instantaneously.

The goal for the thesis at hand is for systems with an untrusted service to achieve a graceful degradation of service semantics (also defined in the notion of a *Byzantine emulation* [CSS07]): As long as the service behaves correctly, user operations are processed atomically, and thus, satisfying linearizability. In any case where the untrusted service deviates from its specified behavior, the *fork-linearizability* property makes sure that each user still observes operations (and their effects on the shared object) in a linearizable order, although operations might have been omitted. This implies that the untrusted server can still fork the views of two users but once forked they will remain forked forever (the so-called *no-join* property). More formally, fork-linearizability guarantees that every two users observe a common prefix of the linearizable order of operations up to the forking point.

Mission Impossible

During the last decade a number of protocols and systems implementing *fork-linearizability* have been published [MS02, LKMS04, CSS07, CG09, Cac11]. All these protocols have in common that they are based on *locks*. The use of *locks* is a well-known and generic principle to serialize the access to a shared object. A lock

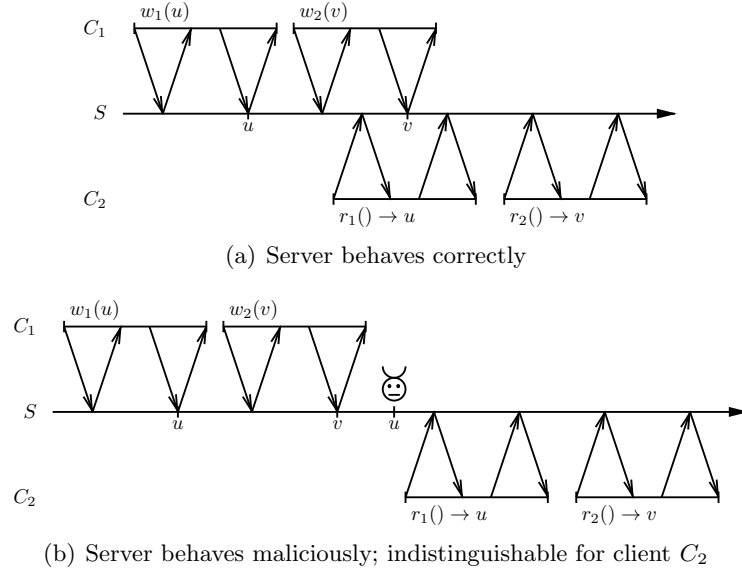
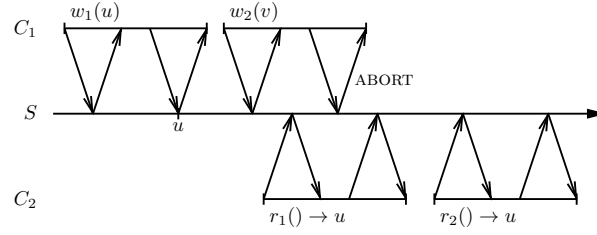


Figure 1.1: Intuition behind Cachin's impossibility result [CSS07]

is a unique data item, which grants the client holding the lock exclusive access to the shared object. The fundamental problem of a lock-based approach is that if the lock is lost, e.g., when the client holding the lock crashes or is permanently disconnected from the network, the whole system is blocked. The system may even block if the untrusted service does not act maliciously. Moreover, it has been found that there exists a fundamental limitation of *fork-linearizable* implementations: Cachin *et al.* have shown [CSS07] that even if the service behaves correctly, clients cannot complete their operations independently from each other as this introduces a vulnerability that can be exploited to violate fork-linearizability. To have operations complete independently of each other is a desirable liveness property called *wait-freedom* [Her91]. It is stronger than the liveness property provided by lock-based systems — a client has to wait for the lock to be released before it can take any action on the shared object, therefore not being independent of the behavior of other clients. In a more formal way as shown by Cachin's impossibility result [CSS07], in an asynchronous system there is no *wait-free* emulation of a fork-linearizable storage on an untrusted server.

To get the intuition behind Cachin's impossibility result, let us assume a simple system with an untrusted server S and two clients C_1 and C_2 . Here, all operations are assumed to be wait-free and complete after two rounds of communication with the server (see Figure 1.1). Figure 1.1(a), where the servers behaves correctly, shows two write operations executed by client C_1 storing values u and v one after

Figure 1.2: Intuition behind *abortable* operations

the other on server S . Read operation r_1 of client C_2 is concurrent with write operation w_2 and is thus allowed to return value u . The second read operation happens after operation w_2 and returns the newer value v . In this example it can be observed that the information sent by C_1 in the second communication round of write operation w_2 does not depend on any action taken by client C_2 . This fact is exploited by the maliciously behaving server in Figure 1.1(b) to violate fork-linearizability. Here, server S omits value v from client C_2 and presents the outdated value u . During read operation r_1 client C_2 is unable to distinguish the real concurrency from Figure 1.1(a) from the simulated concurrency in Figure 1.1(b). Hence, if operations are wait-free fork-linearizability may be violated.

How to Get Around

The first contribution of the thesis at hand is to show how the impossibility result can be circumvented without using a lock-based approach. The main idea to prevent a case like in Figure 1.1 is to allow operations to *abort*. Aborting means that an operation completes by returning the special value **ABORT** and without having an effect to the shared server. Obviously, any protocol implementing *abortable* [AFH⁺07] operations does not satisfy wait-freedom. Instead, the weaker progress condition *obstruction-freedom* [HLM03] is ensured. Building upon the idea of abortable operation the first contribution of this thesis constitutes a protocol implementing a **fork-linearizable shared memory with abortable operations (C1)**. It circumvents the impossibility result by sacrificing the *wait-freedom* progress condition for guaranteeing *fork-linearizability*. As mentioned before, key of this protocol is to avoid a situation like in the example in Figure 1.1(a). In this case, the proposed algorithm *aborts* write operation w_2 (Figure 1.2) such that it does not interfere with read operation r_1 anymore and *fork-linearizability* is not violated.

There are three *lock-free*¹ progress conditions that have been extensively stud-

¹In some papers the term *lock-free* is used synonymously for *non-blocking*. However, in the remainder of the thesis these two notions are used as proposed by Valois [Val94] — distin-

ied (in decreasing order of strength): wait-freedom, non-blocking, and obstruction-freedom. Differently from wait-freedom [Her91], the *non-blocking* progress condition ensures that under concurrency, at least one operation is able to complete [HW90]. Finally, *obstruction-freedom* [HLM03] guarantees that each operation that runs in isolation eventually completes. In practical settings the differences between these three progress conditions have shown to be vincible making obstruction-free protocols very attractive. It has been shown that abortable implementations (and therefore also obstruction-free ones) can be boosted to wait-freedom in practical systems [AT08, FLMS05] that are partially synchronous.

The protocol proposed as the first contribution of this thesis implements a shared service, termed *shared memory*. It consists of n simple storage objects known as single-writer multiple-reader (SWMR) registers. There are n clients in the system, i.e., the total number of readers and writers. Each register of the shared memory provides a READ operation that may be called by any client and a WRITE operation that can only be invoked by a dedicated client. By this, each client can exclusively write into its “own” register and read from all other registers. Therefore, the implemented operations of the shared memory take an additional argument indicating the register of the memory that shall be accessed: $\text{WRITE}(v, i)$ denotes that value v is written into register with index i , while $\text{READ}(i)$ returns the most current content from register with index i . Whenever two of these operations of the shared memory are called concurrently, i.e., the one is invoked while the other has not yet returned, the pending operation is *aborted*. Thereby, contribution (C1) constitutes the first *lock-free* fork-linearizable implementation at all. Moreover, the implemented fork-linearizable shared memory with abortable operations (C1) [MDSS09] offers a very low communication complexity, i.e., the complexity of the length of the sent messages, of $\mathcal{O}(n)$.

Better is the Enemy of Good

Referring back to Figure 1.1(b) let us now have a closer look to why *fork-linearizability* is violated in this example. Here, all operations are executed sequentially in the following order, called real-time order: w_1, w_2, r_1, r_2 . However, as read operation r_1 returns u , the value written by operation w_1 , r_1 must not be ordered *before* w_2 according to the sequential specification. This implies that no order of the operations exists satisfying both, the real-time order and the sequential specification, and thus fork-linearizability is violated. Hence, the problem in this situation (i.e., why no order satisfying fork-linearizability can be found) arises from the fact that operations w_2 and r_1 are not completely independent — r_1 has to be ordered before w_2 and after w_1 . However, considering the operations provided by

guishing protocols that are not based on any locking mechanism (i.e., *lock-free* ones) from protocols that satisfy the *non-blocking* progress condition.

the fork-linearizable shared memory implementation of contribution (C1), there may exist a run of the protocol where concurrent operations are *independent*. As an example, we consider two operations $\text{WRITE}(v, i)$ and $\text{READ}(j)$ where $i \neq j$. Note, that both operations access different registers of the shared memory, and hence, the sequential specification does not dictate any order on them. Thus, for such operations the argumentation of Cachin’s impossibility result [CSS07] does not apply anymore. This further means, that aborting, as done by the protocol of contribution (C1), is unnecessary in case $\text{WRITE}(v, i)$ and $\text{READ}(j)$ are concurrent. In general, concurrent operations need not to be aborted if they access different registers of the shared memory.

The second contribution of this thesis improves on the handling of concurrent operations and reduces the number of unnecessarily aborted operations in comparison to the protocol of contribution (C1). The second contribution of the thesis at hand comprises a protocol implementing a **fork-linearizable shared memory with abortable operations, where concurrent operations accessing different registers are not aborted (C2)**. The protocol of contribution (C2) is an improvement over the one presented in contribution (C1) as in any run of the protocol from (C2) as many as or *more* operations successfully complete than if the same run was executed by the protocol from (C1).

However, the second protocol achieves this improvement by a higher communication complexity of $\mathcal{O}(n^2)$ (where n is the total number of clients that also equals the number of registers of the shared memory). Intuitively, the reason for this is the following: To serialize *all* operations, the protocol of contribution (C1) makes use of a vector of timestamps (of length n) that is digitally signed and appended to each value written into the shared memory. As the second protocol allows for concurrent operations if they are applied to distinct registers, only operations on the same register are serialized. Hence, we apply the “serialization” mechanism based on a timestamp vector of length n to each of the n registers of the shared memory which results in a *timestamp matrix* of size $n \times n$. This in turn leads to the communication complexity of $\mathcal{O}(n^2)$ as such a matrix is assigned to each operation.

In summary, the two protocols of contribution (C1) and (C2) are the first implementations that ensure fork-linearizable semantics while not being based on locks. These protocols exhibit different characteristics: The first one is very efficient given its linear communication complexity while the second one reduces the number of aborted operations arguably to a minimum. As long as the untrusted service (i.e., the shared memory in this case) behaves correctly, both protocols ensure linearizability and obstruction-freedom. If the service deviates from its desired behavior, the safety property of the properties gracefully degrades such that *fork-linearizability* is never violated.

Make it Easy

After having introduced the first two contributions of this thesis, we will now have a closer look at the assumptions underlying the two presented protocols and other approaches in the research area of fork-consistency. The two protocols from contribution (C1) and (C2) both implement a shared memory, a service providing a quite simple interface of just two operations `READ` and `WRITE`. On the other hand, however, the shared service itself, underlying these protocols, requires the execution of non-trivial computation steps. The same observation can be made for existing works: fork-consistent constructions of shared memory [CKS11, SCC⁺10] and file systems [MS02, LKMS04] are based on services that offer stronger operations than a simple read-write interface. In other words, up to now, all fork-consistent emulation protocols have required the service to execute non-trivial computation steps, i.e., the service must implement an object of *universal* type [Her91], capable of *read-modify-write* operations [KRS88].

Such objects, underlying the shared service, are usually classified according to Herlihy's wait-free hierarchy [Her91]. This hierarchy classifies shared objects along their ability to solve the *consensus* problem in a wait-free manner. Consensus can be seen as a fundamental building block to construct a shared object with any functionality (i.e., a universal object) that can be accessed concurrently. The strongest objects in Herlihy's wait-free hierarchy are *read-modify-write* objects, also constituting the universal object type, as they are able to solve *consensus* in a *wait-free* manner with an *arbitrary* number of clients. The weakest kind of objects in this hierarchy are *registers*, providing only read and write operations to the clients. Using a register only the trivial instance of consensus with a *single* client achieves wait-freedom. That is why the so called *consensus number* of registers equals 1, while the universal objects have an infinite *consensus number*. However, it is important to point out that there exists a fundamental difference between the quite weak registers and objects of universal type. This fundamental difference has also been described by Taubenfeld's *power number* [Tau09].

Having this in mind, the requirements of the above mentioned fork-consistent implementation tend to appear counterintuitive: They rely on the strongest possible object type while implementing a weaker one. Hence, the fundamental question comes up whether one can provide a fork-consistent emulation in which the service does not execute computation steps (i.e., constituting a universal object), but can be realized only by memory objects (i.e., registers)? Surprisingly, the next contributions of the thesis at hand show that this question can be answered in the affirmative. Contributions (C3) and (C4) imply that for a wide range of fork-consistent emulations, the underlying universal objects can be replaced by registers. In practical terms, it is important to reduce the complexity and cost of a shared service implementation as computation resources are typically more

expensive than storage.

Also in theory, a long tradition of research has already addressed how to realize more powerful abstractions from weaker base objects (e.g., [Her91, AFH⁺07, AKMS11]). Aguilera *et al.* [AFH⁺07] showed how to construct a shared object with universal functionality from simple register objects. Obviously, their construction does not achieve wait-free operations, as this would contradict the classification of registers and universal objects in Herlihy’s wait-free hierarchy [Her91]. Instead, Aguilera *et al.* [AFH⁺07] construct a universal object with *abortable* operations and argue that the achieved progress condition is as strong as *obstruction-freedom*. So far, no stronger progress condition has been achieved for a construction of a universal object from registers.

There Ain’t No Such Thing As a Free Lunch

Referring back to Cachin’s impossibility result [CSS07], there is no wait-free implementation of a shared memory that satisfies fork-linearizability. Needless to say, this impossibility also applies for implementations built from registers. As fork-linearizability and wait-freedom are the desired properties, any solution constitutes a trade-off sacrificing either the safety or the liveness side of the properties. However, implementations of a universal shared object using only registers are, even without fork-linearizable semantics, restricted to a progress condition weaker than wait-freedom — the strongest known implementation is only obstruction-free. Hence, if there was an implementation which is based only on registers and satisfies fork-linearizability, it would not sacrifice any property in comparison to its counterpart without fork-linearizable semantics.

This goal is achieved by contribution (C3) constituting a **fork-linearizable implementation of a universal object from registers where operations are abortable (C3)**. In comparison to the universal object construction from registers of Aguilera [AFH⁺07], the implementation of contribution (C3) shows that fork-linearizability can be added to such an implementation without being faced with new trade-offs. Moreover, contribution (C3) constitutes the first construction of a universal object with fork-linearizability only from registers. The only existing fork-linearizable universal object implementation is based on a server that implements a universal object [Cac11]. In comparison to further existing fork-linearizability implementations, contribution (C3) allows to replace the server used in contribution (C1) and (C2) by one that implements only weaker register objects. Analogously, existing fork-linearizable services that are based on a server executing non-trivial computation steps and that use *locks* may be implemented with the universal construction from contribution (C3) only from registers *and* without using locks (as operations are abortable). Works from this category are the fork-linearizable file system SUNDR of Mazières and Shasha [MS02], and the

fork-linearizable memory implementation of Cachin *et al.* [CSS07].

To implement the protocol of contribution (C3) a number of *registers* is required that is linear in the number of clients (in total there are n clients). The implementation from contribution (C3), as well as the protocol from (C1) and the one of Cachin [CSS07], makes use of vector of timestamps to ensure fork-linearizability. Unlike the latter protocols, contribution (C3) exhibits a communication complexity of $O(n^2)$. This however is a direct consequence of using a server that implements only registers instead of computationally stronger one: While the latter one is able to return the most recent data on request, in a register based implementation this is impossible. Here, the client has to read from *all* n registers on the server to find the most recent data and thereby increasing communication complexity by factor of n .

A Good Deal

Up to now, all fork-consistent implementations discussed in the thesis at hand have sacrificed wait-freedom to enable fork-linearizability as satisfying both is ruled out by Cachin’s impossibility result [CSS07]. Given the fact that for practical applications obstruction-freedom is also a desirable property that can be easily boosted to wait-freedom [AT08], these approaches turn out to be a good deal. However, there exists also another possibility to circumvent Cachin’s impossibility result by keeping wait-freedom and trading fork-linearizability.

As mentioned earlier, *fork-linearizability* is the strongest fork-consistent property, being based on linearizability. A property weaker than linearizability is *sequential consistency* that gives up the property of *real-time order* required for linearizability. There also exists a fork-consistent counterpart, called *fork-sequential consistency* [OR06] that is strictly weaker than *fork-linearizability*. However, in a recent paper, Cachin *et al.* [CKS09a] have shown that there is no wait-free implementation of fork-sequential consistent storage as well. Hence, targeting fork-sequential consistent approaches does not help in getting around the wait-free impossibility [CSS07].

A possibly viable solution comes in terms of the notion of *weak fork-linearizability* [CKS11] which constitutes the strongest fork-consistent property that allows for wait-free operations. It is strictly weaker than *fork-linearizability* but neither stronger nor weaker than *fork-sequential consistency*. It relaxes *fork-linearizability* in two ways: *Weak fork-linearizability* allows two clients, after being forked, to observe a single operation of the other one (at-most-one-join), and that the real-time order induced by linearizability may be violated by the last operation of each client (weak real-time order). Although weaker than fork-linearizability, *weak fork-linearizability* is also very attractive as it has shown to be of practical relevance: The storage service FAUST [CKS11] achieves weak fork-linearizability.

	Implemented Functionality	Fork-Consistent Property	Liveness with correct server	Server type
(C1)	Shared Memory	Fork- Linearizability	Obstruction- freedom	Universal
(C2)	Shared Memory	Fork- Linearizability	Obstruction- freedom	Universal
(C3)	Universal Object	Fork- Linearizability	Obstruction- freedom	Register
(C4)	Shared Memory	Weak Fork- Linearizability	Wait- freedom	Register

Figure 1.3: Overview on Contributions (C1) – (C4)

The Venus system [SCC⁺10] implements the mechanisms behind FAUST and describes a practical solution for ensuring integrity and consistency to the users of cloud storage.

Contribution (C4) strikes the path of trading fork-linearizability to achieve wait-freedom: It constitutes a **weak fork-linearizable implementation of a shared memory from registers where operations are wait-free (C4)**. Contribution (C4) improves over the only existing weak fork-linearizable construction of a shared memory [CKS11] as it is only based on registers instead of a stronger server implementing a universal object — i.e., the construction from (C4) allows to eliminate the server code from Venus [SCC⁺10]. However, directly implied by using a server that implements only registers is the fact that the implementation from contribution (C4) requires two rounds of communication of the clients with the server while the FAUST service [CKS11] requires only a single round. The communication complexity of $O(n^3)$ leaves room for further improvements. However, the main benefit of the construction of contribution (C4) is that the implemented operations are wait-free, which is the only progress condition guaranteeing maximal independence among client operations [HS11].

As a conclusion, the protocols from contributions (C3) and (C4) [MDCS11] constitute the first known result that fork-consistent semantics can be implemented only from registers. Both focus on different aspects to circumvent existing impossibility results: The protocol from contribution (C3) satisfies fork-linearizability and implements a shared object of universal type. Similar to non-fork-consistent universal constructions from registers, operations may abort under concurrency. Hence, fork-linearizability may be “added” to such protocols without making additional assumptions. The protocol from contribution (C4) implements a shared memory object that ensures *weak* fork-linearizability and where operations are wait-free as long as the base registers behave correctly. Weak fork-linearizability is the strongest known fork-consistency property that may be implemented in a

wait-free manner. Moreover, it shows for the first time that registers are sufficient to implement a fork-consistent shared memory. An overview on contributions (C1) to (C4) of the thesis at hand is given in Figure 1.3 summarizing all introduced protocols with fork-consistent semantics.

Going Deeper

Contributions (C3) and (C4) comprise protocols that are built upon atomic registers. So far, these atomic registers are handled as being untrusted, i.e., the registers are allowed to arbitrarily deviate from the specified behavior. However, such worst-case behavior should be an exception and in normal-case the atomic registers are expected to act as specified. Thus, an obvious question is how to implement the used atomic registers such that the influence of malicious behavior is mitigated? The idea is to find a Byzantine fault-tolerant implementation of an atomic register over a set of replicated servers. The principle of replication allows to tolerate arbitrary behavior of a fraction of the replicas and, thereby, to *mask* [AS85] malicious behavior.

A *register* is a simple storage object that may store only one value at a time and that provides operations READ and WRITE at its interface. Registers come at different strengths dependent upon their behavior under concurrent access [Lam86]. For a *safe* register, which is the weakest register type, holds that a READ operation that occurs concurrently with some WRITE operation may return an arbitrary value. A *regular* register provides stronger properties than a safe register, as a READ operation either returns the value written by the last preceding WRITE or a value which is written concurrently with the READ operation. The strongest, *atomic* register, linearizes all READ and WRITE operations: Additionally to the properties of a regular register it ensure that if a READ operation returns a value v then any succeeding READ does not return a value older than v .

For practical settings it is of vital importance to keep the costs incurred by replication minimal — i.e., to find the maximal number of malicious servers that can be tolerated given any number of replicas. Such an implementation is also denoted as *optimally resilient*. In the context of fault-tolerant storage, a replicated implementation is denoted as *robust* [ABND95] if it is optimally resilient and provides wait-free operations. The provided resilience depends on the underlying fault-model. For the most general fault-model, allowing Byzantine behavior of the replicated servers and which is also applied here, using $3t + 1$ replicas to tolerate t Byzantine failures has shown to be optimal [MAD02].

Besides low replication costs, as provided by *robust* atomic register implementations, for practical settings further aspects have also to be taken into account to achieve a *lightweight* solution: To attain low latency, the number of communication rounds between the client and the replicated servers has to be minimal. To

avoid the costs of setting-up and operating a public-key cryptosystem, an implementation not relying on self-verifying data is desirable (i.e., non-authenticated data model). For reasons of scalability, an unbounded number of readers should be supported. Furthermore, any number of malicious readers should be tolerated.

Although the research area of fault-tolerant storage has been extensively explored in the past decades, none of the existing solutions is able to satisfy all requirements from the above established “wish list”. For optimally resilient Byzantine fault-tolerant storage *both* READ and WRITE operations having a worst-case latency of a single communication round have shown to be impossible: Work of Abraham *et al.* [ACKM06] proves that any Byzantine fault-tolerant storage employing the optimal number of replicas has at least some WRITE operation completing in two communication rounds. Work of Guerraoui and Vukolic [GV06] rules out reading in a single round even from robust *safe* registers. By allowing the existence of *secret tokens*, Dobre *et al.* [DMSS09] circumvent this impossibility result only for robust *regular* registers. Solutions allowing single-round READ and WRITE operations of *atomic* storage are only optimized for the best case [GLV06], i.e., for synchronous runs without concurrency and with fewer malicious replicas. In such runs, the latency of READ and WRITE operations gracefully degrades to two or more communication rounds [GV07], depending on the number of malicious replicas. To finalize the “wish list”, aiming at a robust atomic register implementation where READ operations complete in two communication rounds is optimal, even for the stronger model of authenticated data [DGLV10].

No One Knows What the Future Holds

Contribution (C5) achieves in the non-authenticated data model all requirements on the “wish list” by slightly restricting the behavior of the malicious servers. As in the system model of Dobre *et al.* [DMSS09], the existence of *secret tokens* is allowed. In a practical setting this assumption is easily satisfiable by a (pseudo) random number generator whose output may not be predicted by the malicious servers. Contribution (C5) leverages this assumption and comprises the first **lightweight Byzantine fault-tolerant, robust implementation of an atomic register**. The implementation is denoted as *lightweight*, as it satisfies optimal resilience, features an optimal latency of two communication rounds for READ and WRITE operations, does not rely on self-verifying data, and supports an unbounded number of malicious readers. Note, that this is the first Byzantine fault-tolerant, robust implementation of an atomic register in the non-authenticated data model that tolerates malicious readers. Moreover, the latency of two communication rounds for READ and WRITE operations achieved by the protocol introduced as contribution (C5) in the *non-authenticated data model* is close to the optimal latency of one round for WRITE and two rounds for READ

operations in the *authenticated data model* [MR98]. The protocol of contribution (C5) thereby significantly improves on the best, existing implementation in the non-authenticated data model featuring two-round WRITE and four-round READ operations [DGM⁺11].

The key idea behind the atomic register implementation of contribution (C5) is the concept of a *commitment scheme* intuitively implementing the functionality of a lockable box: The writer generates a *secret token* that is neither known nor may be predicted by the replicated servers and the readers. Then, the token is locked in a box by the writer constituting a *commitment*. Just from the locked box (i.e., the commitment) it is impossible to determine the contained token. However, if the writer sends an *opening* revealing the token, it can be easily verified if commitment and token correspond. For the implementation of the commitment-scheme, contribution (C5) gives two options: The use of a cryptographic hash function [RS04] or a variant of Shamir’s secret sharing scheme [Sha79]. The second option makes the proposed atomic register implementation even information theoretically secure [LCAA07, AACL07].

The commitment scheme is employed by contribution (C5) to reduce the latency of atomic READ operations in the non-authenticated data model from *four* to *two* communication rounds. As a generic technique, an *atomic* READ operation can be implemented from *regular* READ and WRITE operations: The key difference between atomic and regular registers is that an atomic READ operation returning value v has to ensure that any succeeding READ operation does not return a value older than v . Hence, an atomic READ can be constructed from a regular READ if the value, that is going to be returned, is written back to the register using operation WRITE. Thus, the existing four-round atomic READ operation can be seen as a two-round regular READ followed by a two-round WRITE-back. The employed commitment scheme allows the protocol of contribution (C5) to perform the WRITE-back in a single communication round (which would result in a three-round latency for the atomic READ operations). The two-round latency of READ operations is achieved in contribution (C5) by the idea to WRITE-back a *list* of possible values to be returned that is already written back before the reader knows which value from the list it is going to return.

The Wheel is Come Full Circle

Within the trend of upcoming services “in the cloud” simple read/write interfaces for cloud storage solutions have become popular. A key-value store (KVS) is the most favored storage abstraction for cloud services providing such a simple interface [DHJ⁺07, MTJ⁺08, ALM⁺10, LM10, CWO⁺11]. A KVS allows concurrent access of several clients to store and retrieve data in the cloud. It usually provides four different operations to the clients [BCE⁺12]: a $\text{PUT}(key, v)$ operation stores

value v under a unique key key at the KVS. Operation $GET(key)$ retrieves the correct value associated with key key from the KVS. The $DELETE(key)$ operation removes the value associated with key key, while operation $LIST()$ returns a list of all keys with associated values.

Due to the obvious similarity between the interface of an atomic register and a KVS, the idea for contribution (C6) is to extend the atomic register from contribution (C5) to implement a **lightweight, robust, Byzantine fault-tolerant key-value store**. Intuitively, operation $PUT(key, v)$ can be emulated as an atomic operation $WRITE(v)$ to a register named key , while $GET(key)$ means to atomically read from “register” key . As key key in a KVS may be accessed by multiple clients to store data, operation $PUT(key, v)$ has to be extended from single-writer to multi-writer capabilities. This achieved by an extra round of communication at the beginning of a $PUT(key, v)$ operation to request the highest timestamp used for key key so far². Operation $DELETE(key)$, may be emulated as an $PUT(key, \perp)$ operation storing special null value \perp with key into the KVS. The $LIST()$ operation is implemented reusing techniques from atomic $READ()$ operations, concurrently accessing all possible keys.

1.2 Contributions in a Nutshell

This section provides an overview on the contributions made in the thesis at hand. Contributions (C1) and (C2) already appear in the following publications:

[MDCS11] Matthias Majuntke, Dan Dobre, Christian Cachin, and Neeraj Suri. *Fork-Consistent Constructions From Registers*. In Proceedings of the 15th International Conference on Principles of Distributed Systems (OPODIS), Toulouse, France, 2011.

[MDS11] Matthias Majuntke, Dan Dobre, and Neeraj Suri. *Fork-Consistent Constructions From Registers* (Brief Announcement). In Proceedings of the 30th annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), San Jose, California, USA, 2011.

Contributions (C3) and (C4) have been published in the following paper:

[MDSS09] Matthias Majuntke, Dan Dobre, Marco Serafini, and Neeraj Suri. *Abortable Fork-Linearizable Storage*. In Proceedings of the 13th International Conference on Principles of Distributed Systems (OPODIS), Nîmes, France, 2009.

²This extra round of communication may be skipped if writers have access to synchronized clocks.

Contributions (C5) and (C6) are part of the following technical report:

[MDS12] Matthias Majuntke, Dan Dobre, and Neeraj Suri. *Lightweight Robust Atomic Storage*. Technical Report. Technische Universität Darmstadt. TR-TUD-DEEDS-07-01-2012. July, 2012.

Contributions (C1)–(C4) have in common that they allow the shared server used to arbitrarily deviate from its specified behavior (also known as *Byzantine* [PSL80] or *non-responsive-arbitrary* [JCT98] faults). As long as the underlying shared server proceeds correctly, the presented protocols achieve linearizability and the specified progress condition. In any other case, the protocols from contributions (C1)–(C4) gracefully degrade to the specified fork-consistent semantics (Figure 1.3 on page 11). As the clients do not trust the Byzantine server, all information sent during the execution of the protocols by each client is *digitally signed* by the corresponding client. The existence of such a cryptographical signature scheme that cannot be compromised by the Byzantine server is assumed. Moreover, the correctness of the protocols is also based on the assumption that the participating clients trust each other — which is also the basis for a reasonable online collaboration.

Contributions (C5) and (C6) are implemented on top of a collection of $3t + 1$ replicated servers, out of which t may be Byzantine faulty. Both implementations rely on the assumption that the writer has access to a (pseudo) random number generator whose output cannot be predicted by the malicious servers and clients. Therefore, the communication channels between writers and correct servers have to be secure and authenticated. Unless no more than t replicas act maliciously, both protocols provide wait-freedom and atomic operations.

The thesis at hand makes the following contributions:

- (C1) The fork-linearizable shared memory protocol LINEAR with abortable operations (Section 1.2.1 and Section 3.5 in Chapter 3).
- (C2) The fork-linearizable shared memory protocol CONCUR with abortable operations that allows concurrent operations (Section 1.2.2 and Section 3.6 in Chapter 3).
- (C3) The fork-linearizable universal object protocol AFL with abortable operations using only registers (Section 1.2.3 and Section 4.4 in Chapter 4).
- (C4) The weak fork-linearizable shared memory protocol WFL with wait-free operations using only registers (Section 1.2.4 and Section 4.5 in Chapter 4).
- (C5) The lightweight, robust, Byzantine fault-tolerant atomic register protocol LWR (Section 1.2.5 and Section 5.4 in Chapter 5).

- (C6) The lightweight, robust, Byzantine fault-tolerant key-value store protocol LwKVS (Section 1.2.6 and Section 5.5 in Chapter 5).

1.2.1 Abortable Fork-Linearizable Storage

The first contribution (cf. (C1), page 5, and Section 3.5 in Chapter 3) of the thesis at hand deals with the problem of implementing a fork-linearizable shared memory which does not make use of *locks*. Implied by the impossibility result of Cachin *et al.* [CSS07], which rules out the existence of a *wait-free* Byzantine emulation of a fork-linearizable shared memory, all existing approaches implementing fork-linearizable semantics are based on locks [MS02, CSS07].

The LINEAR protocol, constituting contribution (C1), implements for n clients C_1, \dots, C_n a fork-linearizable shared memory with *abortable* operations [AFH⁺07] using a shared server. The implemented shared memory provides n storage registers to the clients, where each client may write to a dedicated register and read from all registers. The shared memory is accessed by two operations $\text{READ}(i)$, that returns the most recently written value from the i th register of the shared memory, and $\text{WRITE}(i, v)$, that updates the state of the i th register with value v and may only be called by client C_i .

The operations READ and WRITE implemented by the LINEAR protocol are *abortable*, i.e., operations are allowed to return the special value ABORT whenever they access the shared memory concurrently. To handle concurrent operations, the shared server implements a concurrency detection mechanism that resembles ideas from recent work of Aguilera *et al.* [AFH⁺07]. During the execution of any READ or WRITE operation, clients communicate with the shared server by message passing. Hence, from the server's point of view an operation is initiated on reception of the first message from the client and finishes as soon as the server has sent the final message of an operation to the client. To detect concurrency among operations, the server maintains a list of *pending* operations, i.e., for which it has received the initial message from the client but not yet sent the final message. Upon reception of the initial message of a new operation, the server relabels all pending operations as "to be aborted" and maintains the new operation as the only pending operation. The final message of an operation that is labeled for abortion, indicates to the client that the operation has not successfully completed and the client may return the special ABORT value. The effects of aborted operations are rolled-back by the server.

Similar to the works of Mazières and Shasha [MS02] and Cachin *et al.* [CSS07] the fork-linearizable semantics are realized in the LINEAR protocol by timestamp vectors. Each READ and WRITE operation is assigned a vector of length n of timestamps which is written to the shared server. The basic principle is that a client upon initialization of an operation requests the most recent timestamp

vector from the server. Next, it performs consistency checks, increments its own timestamp (i.e., client C_i updates the i th entry) in the vector, and sends the updated timestamp vector back to the server. To perform the consistency check, the client always stores the timestamp vector assigned to its last successful operation (i.e., one that has not been aborted). The consistency check is passed (for client C_i) if the i th entry in the timestamp vector returned from the server is not smaller than the i th entry in the timestamp vector of the last successful operation of C_i . It is important to note that all the information exchanged between client and server is *digitally signed* by the client. Therefore, the performed consistency check ensures fork-linearizability as, intuitively, a client may detect if some of its previous operations have been omitted by the server from other clients.

Combining the ideas of concurrency detection to *abort* interleaving operations with timestamp vector based consistency checks to guarantee fork-linearizability, the presented LINEAR protocol constitutes the first *lock-free* implementation of fork-linearizable shared memory. Its communication complexity of $O(n)$ is as low as the complexity of the most-efficient lock-based fork-linearizable emulation [CSS07].

1.2.2 Concurrent Abortable Fork-Linearizable Storage

The second contribution (cf. (C2), page 7, and Section 3.6 in Chapter 3) comprises the CONCUR protocol that improves the handling of concurrent operations compared to the LINEAR protocol of contribution (C1). The CONCUR protocol as well constructs a fork-linearizable shared memory for n clients with abortable operations. It implements the same interface as the LINEAR protocol providing operations $\text{READ}(i)$ and $\text{WRITE}(i, v)$ to each client C_i , $i = 1, \dots, n$. In contrast to the LINEAR protocol, the presented CONCUR protocol does not abort concurrent operations as long as they access different registers of the shared memory, e.g., operation $\text{WRITE}(i, v)$ writing value v into the i th register and operation $\text{READ}(j)$ that concurrently reads from the j th register.

However, this implies that the CONCUR protocol has to be able to cope with concurrent operations: The basic principle of reading, updating, and writing timestamp vectors to ensure fork-linearizability is adopted from the LINEAR protocol. By allowing concurrent operations, a situation may arise where the server returns the same timestamp vector to two different operations of clients C_i and C_j . Here, client C_i would increase the i th entry in the timestamp vector while C_j increases the j th entry. Hence, neither the new timestamp vector of C_i would pass the next consistency check of client C_j nor vice versa. To overcome this problem, the CONCUR protocol applies the timestamp vector scheme from the LINEAR protocol separately to each of the n registers. Thus, each operation is now assigned n timestamp vectors that can be arranged as the rows of an $n \times n$ matrix.

Relying on a timestamp matrix rather than a timestamp vector, the communication complexity of the CONCUR protocol increases to $O(n^2)$. In comparison to the LINEAR protocol, however, a higher number of operations complete without being aborted in the CONCUR protocol. In a direct comparison, there exists no run where the CONCUR protocol aborts an operation while the LINEAR protocol does not. To implement the refined concurrency detection mechanism, the CONCUR protocol maintains a separate list of pending operations for each register of the shared memory. Hence, the CONCUR protocol is the first fork-linearizable implementation of a shared memory that allows operations to be successfully executed concurrently.

1.2.3 An Abortable Fork-Linearizable Universal Object from Registers

The third contribution (cf. (C3), page 9, and Section 4.4 in Chapter 4) of this thesis reduces the assumptions made in comparison to the LINEAR and CONCUR protocols from contribution (C1) and (C2). The presented protocol AFL implements a shared object of universal type only from registers, that satisfies fork-linearizability and where operations are allowed to abort under concurrency.

The basic idea of the AFL protocol follows the construction of a universal object from registers developed by Aguilera *et al.* [AFH⁺07]: To implement an arbitrary functionality, the client reads the most recent state stored in the registers at the shared server, applies the corresponding state transformation, and writes the new state into its “own” register (i.e., client C_i writes to the i th register). Before, a state is written to a register on the server, the assigned timestamp vector has to pass a consistency check, analogously as in the LINEAR protocol.

Similar to the LINEAR protocol, the AFL protocol makes use of timestamp vectors to ensure fork-linearizability and implements a concurrency detection mechanism to determine operations that have to be aborted. As the used registers offer a very limited functionality in comparison to the shared server underlying contributions (C1) and (C2), the concurrency detection has to be changed fundamentally. Therefore, the AFL protocol uses a special INC&READ *counter* object C . This counter object C is a wait-free variant of the one proposed by Aguilera *et al.* [AFH⁺07] to construct a universal object with abortable operations from registers. The INC&READ counter offers two operations: READ(C) that returns the latest counter value, and INC&READ(C) that increments the counter and returns the new counter value in an atomic operation. The operations of the INC&READ counter object C are wait-free and it can be constructed from n registers.

The AFL protocol uses the INC&READ counter for two different purposes, to detect concurrent operations and to determine the most recent state of the implemented universal object. To detect concurrent operations, the client calls INC&READ(C) upon initiation of an operation and stores the returned counter

value. Before completing an operation, the client reads from the INC&READ counter (using $\text{READ}(C)$) and compares the new counter value with the stored one. If it has changed, another client has incremented the counter in the meanwhile, implying that the operation is under concurrency and has to be aborted. Furthermore, the returned counter value of C is written together with the updated state of the universal object, which allows the client to determine the state with the highest assigned counter value as the most recent.

The proposed AFL protocol is the first construction of a universal object from registers that features fork-linearizable semantics. All previous implementations of fork-linearizability are based on computationally stronger servers [MS02, CSS07, MDSS09]. In comparison to these works, the AFL protocol, by requiring only registers, achieves a significant reduction of the underlying assumptions. Compared to the only other existing construction of a universal object from registers with abortable operations [AFH⁺07], the AFL protocol additionally achieves fork-linearizability without making further assumptions.

1.2.4 Wait-free Weak-Fork-Linearizable Storage from Registers

The forth contribution (cf. (C4), page 11, and Section 4.5 in Chapter 4) of this thesis comprises the WFL protocol implementing a shared memory only using registers. In contrast to the AFL protocol, the operations of the WFL protocol achieve wait-freedom as long as the underlying server providing the shared registers behaves correctly. Induced by Cachin’s impossibility result [CSS07], the WFL protocol implements *weak fork-linearizability*.

The proposed WFL protocol makes use of an atomic single-writer snapshot object S with n components [AGR08, Fic05]. Snapshot object S provides two atomic operations: $\text{UPDATE}(d, S, i)$, that writes d to component i of S , and $\text{SCAN}(S)$ that returns the most recent content of all n components. It has been shown, that snapshot object S can be implemented only from registers in a wait-free manner [AGR08, Fic05]. Moreover, the WFL protocol maintains one dedicated register W_i for each client C_i , $i = 1, \dots, n$.

The basic principle behind the WFL protocol is that each client maintains a local timestamp and that during each operation, first, this timestamp is written to the shared memory, and secondly, the timestamps left by other operations are read. Thereby, during READ operations, client C_i first writes its timestamp tsr by calling $\text{UPDATE}(tsr, S, i)$ and then reads all timestamps from registers W_1, \dots, W_n . During a WRITE operation, client C_i first writes its timestamp to register W_i before it reads the timestamps of other clients using operation $\text{SCAN}(S)$. This guarantees a helpful property on the implemented READ and WRITE operations: Whenever read- and write-phase of two operations are concurrent, the corresponding UPDATE and SCAN phases are not concurrent (Figure 1.4).

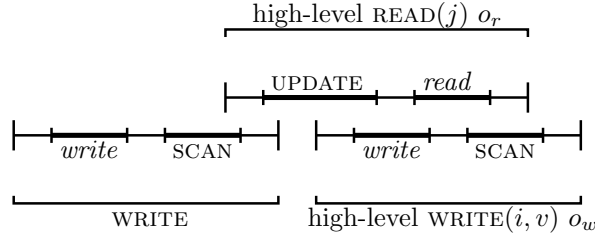


Figure 1.4: Basic principle implemented by the WFL protocol

In the example of Figure 1.4, the call of SCAN during write operation o_w will return the timestamp written during read operation o_r using UPDATE — hence, operation o_w “has seen” operation o_r . Weak fork-linearizability is guaranteed in the WFL protocol by having the clients write such information during each WRITE operation. During READ operations a consistency check is performed probing whether a WRITE operation “has seen” the expected set of READ operations.

The WFL protocol constitutes the first weak fork-linearizable construction of a shared memory using only registers. In this sense it achieves the same properties as the FAUST service of Cachin *et al.* [CKS11] (which is based on a server with universal functionality), but makes significantly less assumptions.

1.2.5 A Lightweight Robust Atomic Register

Contribution (C5) (page 13, and Section 5.4 in Chapter 5) of this thesis comprises a lightweight and robust Byzantine fault-tolerant implementation of a SWMR atomic register. Robustness denotes that the proposed implementation provides wait-free operations and is optimally resilient with respect to Byzantine failures, i.e., it tolerates t malicious out of $n = 3t + 1$ replicas [MAD02]. The provided READ and WRITE operations feature a worst-case latency of two communication rounds which has shown to be optimal for READ operations even in settings with self-verifying data [DGLV10].

The implemented atomic register has been designed to be lightweight under several practical aspects: it does not make use of self-verifying data to avoid the costs incurred by a public-key cryptosystem (non-authenticated data model). It is scalable in the number of clients as the communication complexity is in $\mathcal{O}(n)$. It tolerates any number of malicious readers.

The basic principle underlying the implemented atomic register is that it makes use of a cryptographic commitment scheme [BCC88]. During the pre-write phase of a WRITE(v) operation, the writer generates a token and sends, along with value v , a *commitment* to the replicated servers. By the *hiding* property of the commitment scheme, the chosen token cannot be known by the replicas. In the write-

phase, the writer sends an *opening* to the servers, revealing the token. The *binding* property of the commitment ensures that the servers can validate the revealed token.

Accessing $t + 1$ replicas reporting a validation of the token indicates to a client during a `READ()` operation that the corresponding pre-write phase is complete. Thus, value v has been stored to $t + 1$ *correct* servers which is a sufficient condition to perform the write-back of v , to ensure atomic semantics, in a single communication round.

The introduced atomic register constitutes the first robust and latency-optimal implementation in the non-authenticated data model where an unbounded number of malicious readers is supported. The used model is based on the assumption that malicious servers may not predict the output of a (pseudo) random number generator [DMSS09].

1.2.6 A Lightweight Robust Key-Value Store

The final contribution of the thesis at hand (contribution (C6), page 15, and Section 5.5 in Chapter 5) extends the SWMR atomic register implementation from contribution (C5) to a Byzantine fault-tolerant *key-value store* (KVS) [BCE⁺12]. As well, the proposed KVS is optimally-resilient with respect to Byzantine faulty objects, supports an unbounded number of clients, and tolerates an unbounded number of malicious readers. Operations `GET(key)` complete in two communication rounds. As the KVS supports multiple writers an extra round of communication is required. Thus, `PUT(key, v)` and `DELETE(key)` operations complete after three rounds of communication. Operations, `GET(key)`, `PUT(key, v)`, and `DELETE(key)` are atomic, while `LIST()` operations feature *regular* semantics.

Contribution (C6) provides the most efficient Byzantine fault-tolerant implementation of a KVS which has become one of the most popular interfaces for “cloud storage” solutions.

2 Preliminaries

This chapter introduces the system model underlying all protocols that are proposed throughout this thesis (Section 2.1). All protocols are executed in an asynchronous distributed system consisting of clients and servers. The protocols implement different shared functionalities that provide operations at the client interfaces to be called by higher-level applications. In the later Chapters 3, 4, and 5 the basic model is extended as required by the corresponding contributions. Section 2.2 builds up a background on different types of servers and on fork-consistency, required for a deeper understanding of the contributions of the thesis at hand. It is differentiated between servers that are able to implement a universal functionality and those that provide only a simple read/write interface. Fork-consistency refers to a class of safety properties that can be achieved in a system with a malicious server. The corresponding liveness properties are only guaranteed as long as the server behaves correctly.

2.1 System Model

2.1.1 Basic Concepts

The system underlying the protocols introduced in the thesis at hand is a distributed system consisting of two sets \mathcal{C} and \mathcal{S} of processes denoted as *clients* and *servers*, respectively. Each process is modeled as a *deterministic I/O automaton* [Lyn98, Sec. 8.1]. Every client may communicate with every server by message passing over communication channels. Servers are unable to communicate with each other. The state of a communication channel between client p and server q is modeled as a set of messages $mbuf_{p,q} = mbuf_{q,p}$.

A distributed protocol \mathcal{P} consists of a collection of algorithms \mathcal{A}^P , where each process p is assigned algorithm A_p from collection \mathcal{A}^P . A process p executes algorithm A_p if it proceeds in *steps* of A_p . A *step* is defined as a pair of process id and a set of messages (p, M) , where set M might also be the empty set \emptyset . A process p *takes* a step (p, M) if it performs the following actions:

1. It removes the messages M from set $mbuf_{p,*}$;
2. it applies M and its current state st_p to its local automaton defined by A_p , which outputs a new state st'_p and a set of messages M_{out} to be sent, and it

adopts st'_p as its new state;

3. it puts messages M_{out} in set $mbuf_{p,*}$.

The servers are assumed to be *passive*¹ [ACKM06], i.e., a server sends messages to clients only in reply to a client message. Formally, server p may add messages to set $mbuf_{p,q}$ in step $s = (p, M)$ only if in step s it received a message $m \in M$ from client q .

A process p that takes steps according to its algorithm A_p is called *non-malicious*. Process p is denoted as *malicious* if it performs arbitrary actions: It may remove and add an arbitrary set of messages to set $mbuf_{p,*}$ and change its state in an arbitrary way. Malicious processes are also called *Byzantine-faulty* [PSL80] as they exhibit non-responsive-arbitrary faults [JCT98]. In any run of distributed protocol \mathcal{P} , if a *non-malicious* process executes an infinite number of steps, it is referred to as being *correct*, if it stops taking steps after a finite number of steps it is called *crash-faulty*.

Messages sent between a correct client and a correct server are eventually received, i.e., communication channels are *reliable*. Formally, for two correct processes p and q , if p adds message m to set $mbuf_{p,q}$, then there is a step (q, M) taken by q such that $m \in M$.

In the thesis at hand, we allow the system to be *asynchronous*, i.e., there are no time bounds on message communication nor on processing speeds of the processes.

2.1.2 Implementation of Shared Objects

Clients and servers executing distributed protocol \mathcal{P} are used to implement a *shared object* of functionality F . Such a shared object of functionality F provides *operations* at the interface of the clients. An operation is defined by two events occurring at the client, denoted as *invocation* and *response*. At *invocation* of an operation at some client p , client p starts taking steps according to algorithm A_p . At the *response* of an operation, client p stops taking steps until the next invocation and returns the corresponding result at its interface.

An *execution* of distributed protocol \mathcal{P} is defined as the (interleaved) sequence of invocation and response events of the implemented functionality F . Every execution induces a *history* which is the sequence of invocations and responses occurring at the clients. We say that a response *matches* an invocation, if both are events of the same operation. An operation is called *complete* in some history, if there exists a matching response to its invocation, else *incomplete*. We assume that at each client a new operation is only invoked after the previous operation has completed. A history consisting only of matching invocation/response pairs is

¹This model is sometimes referred to as *data centric* [MR00].

called *well-formed*. Operation o *precedes* operation o' in a history σ ($o <_{\sigma} o'$) iff o is complete and the response of o happens before the invocation of o' . If o precedes o' we denote o and o' as *sequential*, if neither one precedes the other, then o and o' are said to be *concurrent*. A sequence of invocation and response events is called *sequential*, if it starts with an invocation and contains only sequential operations.

The implemented functionality F is defined by a *sequential specification*, which defines the allowed behavior if all operations are *sequential*. However, the history of an execution of distributed protocol \mathcal{P} is usually not sequential. We introduce the following consistency condition, *linearizability* [HW90], in Definition 2.2 to define what we mean by saying that \mathcal{P} *implements* functionality F .

Definition 2.1 (Real-time Order). Let σ be a sequence of invocation and response events. A permutation π of σ *preserves the real-time order of σ* if for every operation o that precedes operation o' in σ , operation o also precedes o' in π .

Definition 2.2 (Linearizability). A history σ of an execution of a distributed protocol \mathcal{P} is *linearizable* with respect to a shared functionality F if σ can be extended to a sequence σ' by adding zero or more response events such that σ' consists only of complete operations, and if there exists a sequential permutation π of σ' such that:

1. π preserves the real-time order of σ' ,
2. the operations in π satisfy the sequential specification of functionality F

Thus, a distributed protocol \mathcal{P} implements a shared object with functionality F , if the history of any execution of \mathcal{P} is *linearizable* with respect to functionality F . A distributed protocol \mathcal{P} satisfies *wait-freedom* [Her91], which is the strongest possible *liveness* property, if each operation that is invoked by a *correct* client eventually completes. Intuitively, a wait-free operation terminates independently of other operations. A distributed protocol \mathcal{P} satisfies *obstruction-freedom* [HLM03] if all provided operations are obstruction-free. An *obstruction-free* operation, invoked by a correct client, eventually completes if it executes in isolation, i.e., it is not concurrent to any other operation.

2.1.3 Sequential Specification of Shared Objects

The distributed protocols introduced in the thesis at hand, implement shared objects with four different functionalities.

Atomic Register An *atomic register* is a shared storage object that provides two operations: READ and WRITE. Operation WRITE takes a value v from set \mathcal{V} as input parameter, also denoted as WRITE(v). Operation READ() has no input parameter and returns a value from set $\mathcal{V} \cup \perp$. In a sequential sequence of operations

that contains no operation $\text{WRITE}(v)$ before operation $\text{READ}()$, operation $\text{READ}()$ returns value \perp . Else, in a sequential sequence of operations where operation $\text{WRITE}(v)$ is the last WRITE operation before operation $\text{READ}()$, operation $\text{READ}()$ returns value v .

The clients providing the operations of an atomic register are partitioned into two subsets, a singleton \mathcal{W} and a set \mathcal{R} such that $\mathcal{R} \cap \mathcal{W} = \emptyset$. The client in \mathcal{W} is called the *writer* while clients in \mathcal{R} are denoted as *readers*. Readers only provide READ operations while the writer provides both operations WRITE and READ . The distributed protocol introduced in Section 5.4 implements a shared object with the functionality of an *atomic register*.

Shared Memory The operations provided by a *shared memory* are READ and WRITE and based on the corresponding operations of an atomic register. Both READ and WRITE operations of a shared memory take the client id as an additional input parameter: For all clients $C_i \in \mathcal{C}$: In a sequential sequence of operations that contains no $\text{WRITE}(i, v)$ operation before operation $\text{READ}(i)$, operation $\text{READ}(i)$ returns value \perp . Else, in a sequential sequence of operations where operation $\text{WRITE}(i, v)$ is the last $\text{WRITE}(i, *)$ operation before operation $\text{READ}(i)$, then $\text{READ}(i)$ returns value v .

For each client $C_i \in \mathcal{C}$ holds that only client C_i provides both operations $\text{WRITE}(i, *)$ and $\text{READ}(i)$ while all other clients only provide operation $\text{READ}(i)$. A shared memory may also be presented as a collection of atomic registers R_i , such that $\text{WRITE}(i, v)$ corresponds to operation $\text{WRITE}(v)$ applied to register R_i (analogously for READ operations). If it is clear from the context, we use both presentations interchangeably. The distributed protocols in Sections 3.5, 3.6, and 4.5 implement a shared memory.

Key-Value Store A *key-value store* (KVS) is a shared storage object that provides the four operations PUT , GET , DELETE and LIST . Operation PUT takes value v from set \mathcal{V} and a key k from set \mathcal{K} as input parameters, operations GET and DELETE take only keys as input parameter. Operation GET returns a value from set $\mathcal{V} \cup \mathbf{fail}$. Operation LIST takes no input parameter and returns a set from power set of \mathcal{K} , i.e., $2^{\mathcal{K}}$.

For all keys $k \in \mathcal{K}$: In a sequential sequence of operations that contains no $\text{PUT}(k, v)$ operation before $\text{GET}(k)$ or if $\text{PUT}(k, v)$ is the last $\text{PUT}(k, *)$ operation before operation $\text{GET}(k)$ and there is a $\text{DELETE}(k)$ between $\text{PUT}(k, v)$ and $\text{GET}(k)$, then operation $\text{GET}(k)$ returns **fail**. In a sequential sequence of operations where operation $\text{PUT}(k, v)$ is the last $\text{PUT}(k, *)$ operation before operation $\text{GET}(k)$ and there is no $\text{DELETE}(k)$ between $\text{PUT}(k, v)$ and $\text{GET}(k)$, then operation $\text{GET}(k)$ returns value v .

In a sequential sequence of operations without any PUT operation before operation LIST(), or if between any PUT(k, v) operation before LIST() and operation LIST() there is a DELETE(k) operation, then LIST() returns \emptyset . Let $K \in 2^K$ be a set of keys. In a sequential sequence of operations, where for all $k \in K$ there exists a PUT(k, v) operation before LIST() such that there is no DELETE(k) operation between PUT(k, v) and LIST(), then LIST() returns set of keys K .

The distributed protocol introduced in Section 5.5 implements a shared object with the functionality of a *key-value store*.

Universal Object A universal object may implement any functionality, hence we say that a universal object is of type T [Her91, AFH⁺07]. Type T of a universal object that can be accessed by clients in \mathcal{C} is defined by a tuple (I, R, S, s_0, δ) where I is a set of *instructions*, R is a set of *responses*, S is a set of *states*, $s_0 \in S$ is the *initial state*, and δ is a relation $\delta \subseteq S \times I \times \mathcal{C} \times S \times R$. Intuitively, a tuple (s, ins, i, s', res) is in δ if client $C_i \in \mathcal{C}$ calls instruction ins to the universal object that is in state s , the object moves to state s' and returns response res . To ease the presentation, the type T of a universal object is encoded in the procedure $\text{APPLY}_T : S \times I \times \mathcal{C} \rightarrow S \times R$. For client $C_i \in \mathcal{C}$, state s and instruction ins , $\text{APPLY}_T(s, ins, i)$ returns (s', res) , where s' is the new state of the universal object and res the response, if (s, ins, i, s', res) is in δ .

A universal object provides only one operation EXECUTE to the clients. Operation EXECUTE(ins) takes instruction ins from set I as input parameter and returns a response from set R . In a sequential sequence of operations that contains only one operation EXECUTE(ins) (of client C_i), operation EXECUTE(ins) returns response res_0 such that $\text{APPLY}_T(s_0, ins, i)$ returns (s, res_0) . In a sequential sequence of operations where operation EXECUTE(ins') (of client C_j) directly follows operation EXECUTE(ins) (of client C_i), operation EXECUTE(ins') returns response res' such that (s, res) has been returned by APPLY_T during EXECUTE(ins), and $\text{APPLY}_T(s, ins', j)$ returns (s', res') .

The distributed protocol introduced in Section 4.4 implements a universal object of type T .

2.1.4 Communication Models

In Section 2.1.1 introducing the basic assumptions underlying the system model of this thesis, it has been specified that clients and servers communicate by sending messages and that each server executes an algorithm within a distributed protocol. For the distributed protocols developed in Chapter 4, we slightly deviate from the basic model to ease the presentation of the distributed protocols.

In Chapter 4, the algorithms executed by the servers are omitted. Instead, we say that a server implements a collection of shared objects. These objects (analo-

gously to the shared objects defined in Section 2.1.2), provide operations that can be accessed by the clients. Thus, instead of sending messages to the servers, the clients directly call the operations provided by the servers to communicate with the servers. In Chapter 4, the servers implement shared objects with the functionality of *atomic registers*, or short, the servers implement atomic registers. Hence, the clients can directly access the provided READ and WRITE operations. As shared registers allow to store and retrieve data, the communication model underlying the distributed protocols of Chapter 4 is also referred to as the *shared memory model* [ABND95]. Hiding the algorithms executed by the servers under the abstraction of shared registers is in line with work of Attiya *et al.* [ABND95]: They prove that it is possible to view the shared memory model as a high-level language for designing distributed protocols in asynchronous message-passing systems.

2.2 Background

2.2.1 Comparing Servers

So far, in this thesis two kind of servers have been introduced: The first type that executes a local algorithm from a distributed protocol, and the second type where the execution of the local algorithm is “abstracted away” and that implements shared atomic registers. In this section, we will see that these two types are fundamentally different. Let us refer to the first kind of servers as the *universal* type while the second is denoted as the *register* type. Work of Herlihy [Her91] has shown that the universal type is strictly stronger in its computational power than the register type.

Registers The notion of *registers* as storage objects has been defined by Lamport [Lam86]. Lamport defines three register types that can be ordered by the strength of the properties they satisfy: *safe*, *regular*, and *atomic* registers, while a *safe* register is the weakest and an *atomic* one is the strongest type. All register types provide READ and WRITE operations. Lamport defines registers as *single-writer multiple-reader (SWMR)* registers, where a dedicated client, called *writer*, may access the register using WRITE and READ operations, while all other clients, referred to as *readers*, may only call the register’s READ operation. SWMR registers can be used as a building block to construct registers that also allow multiple writers [IS92]. All registers satisfy the same sequential specification (cf. Section 2.1.3).

Safe, regular, and atomic registers differ in their behavior under concurrency. A READ operation of a *safe* register that is concurrent with a WRITE operation is allowed to return any value from \mathcal{V} . A *regular* register provides stronger consistency,

as a READ operation returns either a value written by a concurrent WRITE operation, or the value written by the latest preceding WRITE operation. An *atomic* register is the strongest type satisfying *linearizability*. Intuitively, this means that additionally to the properties of a *regular* register, a READ operation never returns an older value than the one returned by the latest preceding READ operation.

Universal Type The used servers of the universal type are able to atomically execute *read-modify-write* operations [KRS88]. Such operations allow a server in one atomic step to (1) read a local variable, and (2) to change it dependent on the variable's current state. Well-known examples of such *read-modify-write* operations are fetch-and-cons, sticky bits, compare-and-swap, memory-to-memory swap, and memory-to-memory copy [Her91]. In the context of this thesis the servers of universal type maintain a *timestamp* variable, that is only updated with larger values, requiring a *conditional write*, that falls in the same class of *read-modify-write* operations [CJS12].

Wait-free Hierarchy Herlihy [Her91] compares in a seminal work different types of shared objects by their ability to solve the consensus problem in a *wait-free* manner. The consensus problem requires a set of clients to agree on a common value among the values that have been proposed by the clients. To achieve a *wait-free* solution of the consensus problem the following properties have to be satisfied [CT96]:

- If a client decides value v , then v has been proposed by some client (Validity).
- No two clients decide different values (Agreement).
- Eventually, every correct client decides on some value (Termination).

Herlihy identifies a shared object that solves the consensus problem in a wait-free manner for n clients as *universal* in a sense that there is a construction using such an object that provides any functionality to n clients. Hence, the idea is to classify shared objects according to their *consensus number*. A consensus number of n means that the shared object is able to solve the consensus problem with n clients in a wait-free manner. Herlihy shows that an *atomic register* has consensus number 1, hence it may solve consensus only for the trivial instance of one client. On the other hand, servers that provide *read-modify-write* operations have an *infinite* consensus number.

In this sense, the *universal* servers underlying the distributed protocols in Chapters 3 and 5 are strictly stronger than the servers implementing *atomic registers*, used in Chapter 4.

2.2.2 Fork-Consistency

Intuitively, fork-consistency formalizes the properties that can be achieved in a distributed system with only malicious servers. Fork-consistency is a collective term for a number of such properties. As malicious servers may simply refuse to respond to client requests, fork-consistent properties comprise only safety but no liveness properties.

The strongest fork-consistent property is called *fork-linearizability*. It was introduced by Mazières and Shasha [MS02] and has later been formalized by Cachin *et al.* [CSS07]. Fork-consistent properties which are weaker than fork-linearizability are *fork-sequential consistency* [OR06], *fork-* consistency* [LM07], and *weak fork-linearizability* [CKS11].

The fork-consistent protocols introduced in this thesis are either *fork-linearizable* or *weak fork-linearizable*. The following definition formalizes the notion of fork-linearizability:

Definition 2.3 (Fork-Linearizability). A history σ of an execution of a distributed protocol \mathcal{P} is *fork-linearizable* with respect to a shared functionality F if and only if for each client $C_i \in \mathcal{C}$, there exists a subsequence σ_i of σ consisting only of completed operations and a sequential permutation π_i of σ_i such that:

1. All completed operations in σ occurring at client C_i are contained in σ_i ; and
2. π_i preserves the real-time order of σ_i ; and
3. the operations of π_i satisfy the sequential specification of functionality F ; and
4. for every operation $o \in \pi_i \cap \pi_j$, the sequence of events that precede o in π_i is the same as the sequence of events that precede o in π_j .

The definition of *weak fork-linearizability* is omitted here. Instead, the reader is referred to Chapter 4. As mentioned earlier, fork-consistent properties are safety properties. The following notion of a *Byzantine emulation* [CSS07] allows us to formally define the safety and liveness properties of fork-consistent protocols.

Definition 2.4. A distributed protocol \mathcal{P} *emulates* a shared object with functionality F on a Byzantine server S with $\{\text{fork}|\text{weak fork}\}$ -linearizability whenever the following conditions hold:

1. If server S is correct, the history of any fair² execution of \mathcal{P} is *linearizable* with respect to functionality F , and
2. the history of any fair execution of \mathcal{P} is $\{\text{fork}|\text{weak fork}\}$ -linearizable with respect to functionality F .

²For a formal definition we refer to standard literature [Lyn98].

Such an emulation is *wait-free* (*obstruction-free* resp.), iff any operation of a correct client is wait-free [Her91] (*obstruction-free* [HLM03] resp.) if server S is correct.

The distributed protocols in Sections 3.5 and 3.6 constitute *obstruction-free, Byzantine emulations* of *shared memory* with *fork-linearizability*. The protocol introduced in Section 4.5 *emulates* a *weak fork-linearizable shared memory* on a Byzantine server in a *wait-free* manner. Section 4.4 presents a distributed protocol that is an *obstruction-free, Byzantine emulation* of a *universal object* with *fork-linearizability*

3 Abortable Fork-Linearizable Storage

This chapter introduces the first two contributions (C1) and (C2) of the thesis at hand. The problem which is addressed in this chapter is how to implement a shared memory in an asynchronous message passing system using a server prone to Byzantine failures.

In such an untrusted environment, although cryptographic techniques can be used to ensure confidentiality and integrity of the data, there is no means to prevent a malicious server from returning obsolete data. Fork-linearizability [MS02] guarantees that if a malicious server hides an update of some client from another client, then these two clients will never see each others' updates again. Fork-linearizability is arguably the strongest consistency property attainable in the presence of a malicious server. Recent work [CSS07] has shown that there is no fork-linearizable shared memory emulation that supports *wait-free* operations. On the positive side, it has been shown that *lock-based* emulations exist [MS02, CSS07]. Lock-based protocols are fragile because they are blocking if clients may crash. The two protocols LINEAR (C1) and CONCUR (C2) introduced in this chapter are the first *lock-free* emulations of fork-linearizable shared memory. With a correct server, both protocols guarantee linearizability and that every operation successfully completes in the absence of concurrency, while concurrent operations terminate by *aborting*. The CONCUR protocol additionally ensures that concurrent operations invoked on different data of the shared memory complete successfully.

The two proposed protocols LINEAR and CONCUR are a major step to ease the usability and the development of systems that ensure fork-consistent semantics. *Abortable*, and therefore *obstruction-free* operations, achieve a degree of independence between clients that is fundamentally different from *lock-based* solutions where client operations are strongly dependent on each other. Such independence is desirable as it supersedes complicated mechanism to synchronize the access to critical sections of shared resources. In practical settings [AT08], obstruction-free solutions can be easily extended to *wait-freedom* which ensures maximal independence between client operations.

Section 3.1 gives an introduction and motivates the proposed approaches. Section 3.2 overviews existing works in the related area of research. Section 3.3 refines the system model introduced in Chapter 2 and Section 3.4 details the properties provided by the proposed protocols. Sections 3.5 and 3.6 introduce the protocols LINEAR and CONCUR that constitute contributions (C1) and (C2) of this thesis

and formally prove the protocols correct. A complexity analysis of the LINEAR and CONCUR protocol is given in Section 3.7 that also concludes Chapter 3.

3.1 Introduction

As already discussed in Chapter 1 the upcoming trend of using online or “cloud” services to store personal data provides many benefits to the users. These services offer full data administration such that a user does not need to care for backups or server maintenance and the data is available on demand. Moreover, most of these services allow shared access which makes online collaboration (multiple users working on the same logical data) based on online services very attractive.

Online collaboration usually assumes that the participating clients trust each other — otherwise there exists no basis for reasonable communication. However, when the shared memory is provided by a third party, clients may not fully trust the service as it may corrupt or leak sensitive data. Cryptographic techniques can prevent unauthorized access to data (confidentiality) and undetectable corruption of the data (integrity). On the other hand, progress and consistency cannot always be guaranteed when the storage service is untrusted. A malicious server may simply refuse to reply to client requests and it can violate linearizability by omitting a recent update of one client and presenting an outdated value to another client. This split brain attack is called *forking* and cannot be prevented. However, once a forking attack is mounted, it can be easily detected using a *fork-linearizable* storage protocol. *Fork-linearizability* [MS02] ensures that once two clients are forked, they never see each others’ updates after that without revealing the server as faulty. Once such a partitioning occurs, the clients stop hearing from each other. A client that has not seen updates from another client for a while can use out-of-band communication (as e.g., phone or e-mail) to find out if the server is misbehaving.

Recent work [CSS07] has shown that even if the server behaves correctly, clients cannot complete their operations independently from each other because this introduces a vulnerability that can be exploited by a Byzantine server to violate fork-linearizability. This means that in an asynchronous system there is no *wait-free* [Her91] emulation of fork-linearizable storage on a Byzantine server. On the positive side, the SUNDRA [MS02] protocol and the concurrent protocol by Cachin *et al.* [CSS07] show the existence of fork-linearizable Byzantine emulations using locks. However, lock-based protocols are problematic as they can block in the presence of faulty clients that crash while holding the lock.

Contributions of Chapter 3 This chapter introduces two *lock-free* emulations of fork-linearizable shared memory on an untrusted server. In runs in which the

server behaves correctly, the proposed protocols **LINEAR** and **CONCUR** — constituting contributions (C1) and (C2) of this thesis — ensure linearizability [HW90], and that each operation executed in the absence of concurrency successfully completes. Under concurrency, operations may complete by aborting. Both protocols emulate a shared memory consisting of n single-writer multiple-reader (SWMR) registers, one for each of the n clients, where register i is updated only by client C_i and may be read by all clients. While both protocols address lock-free fork-linearizability, they solve two distinct issues. The **LINEAR** protocol, which is the first *lock-free* fork-linearizable implementation at all, offers a communication complexity of $\mathcal{O}(n)$. The **CONCUR** protocol improves on the handling of concurrent operations such that overlapping operations accessing *different* registers are not perceived as concurrent, and therefore they are not aborted. However, it has a communication complexity of $\mathcal{O}(n^2)$. Both protocols allow concurrent operations to abort in order to circumvent the impossibility result by Cachin *et al.* [CSS07]. The necessary condition for aborting is step contention [AGHK09], and thus, pending operations of crashed clients never cause other operations to abort.

The very basic idea We now give a rough intuition of why aborting helps to circumvent the given impossibility of wait-free fork-linearizability. With both our protocols, if multiple operations compete for the same register, then there is only one winner and all other operations are aborted. On a correct server, this strategy ensures that all successful operations applied to the same register access the register sequentially. Operations have timestamps attached to them and the sequential execution establishes a total order on operations and the corresponding timestamps. The algorithm ensures that a forking attack breaks the total order on timestamps. If a malicious server does not present the most recent update to a READ operation, then the timestamp of the omitted WRITE operation and the one of the READ operation become incomparable — the two clients are forked. The algorithm guarantees that also future operations of those two clients cannot be ordered and thus they remain forked forever.

3.2 Related Work

Mazières and Shasha [MS02] have introduced the notion of fork-linearizability and they have implemented the first fork-linearizable multi-user network file system **SUNDR**. The **SUNDR** protocol may block in case a client crashes even when the storage server is correct. Cachin *et al.* [CSS07] implements a more efficient fork-linearizable storage protocol based on **SUNDR** which reduces communication complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$. The presented protocols are blocking and thus they have the same fundamental drawback as **SUNDR**. The authors [CSS07] also

prove that there is no wait-free emulation of fork-linearizable storage. They do so by exhibiting a run with concurrent operations where some client has to wait for another client to complete. Oprea and Reiter [OR06] define the weaker notion of fork-sequential consistency. Intuitively the difference to fork-linearizability is that fork-sequential consistency does not necessarily preserve the real-time order of operations from different clients. In a recent work, Cachin *et al.* [CKS09a] show that there is no wait-free emulation of fork-sequential consistent storage on a Byzantine server. It is important to note that these impossibility results do not rule out the existence of emulations of fork-linearizable storage with abortable operations [AFH⁺07] or weaker liveness guarantees such as obstruction-freedom [HLM03]. Cachin *et al.* [CKS11] presents the storage service FAUST which wait-free emulates a shared memory with a new consistency semantics called *weak fork-linearizability* (cf. also contribution (C4) in Section 4.5 of Chapter 4). The notion of weak fork-linearizability weakens fork-linearizability in two fundamental ways. After being forked, two clients may see each others' updates once (at-most-on-join property) and secondly, the real-time order among the operations which are the last of each client is not ensured.

Li and Mazières [LM07] study systems where storage is implemented from $3t + 1$ replicated servers and more than t replicas are Byzantine faulty. They present a storage protocol which ensures *fork* consistency*. Similar to weak fork-linearizability, fork* consistency allows that two forked clients may be joined at most once (at-most-one-join property).

The notion of abortable objects has been introduced by recent work of Aguilera *et al.* [AFH⁺07]. The paper shows the existence of a universal abortable object construction from abortable registers. It is the first construction of an obstruction-free universal type from base objects weaker than registers. In a follow-up paper [AT08] it has been shown that in a partially synchronous system, abortable objects can be boosted to wait-free objects. This makes abortable objects, including our abortable fork-linearizable shared memory emulation very attractive.

Summing up, there is no *lock-free* emulation of fork-linearizable storage even though lock-free (i.e., obstruction-free) solutions can be made practically wait-free using boosting techniques as described by Aguilera *et al.* [AT08].

3.3 System Model and Definitions

The system model used in this chapter is based on the system model given in Chapter 2. Here the set of servers \mathcal{S} is a singleton containing only server S . There are n clients in set \mathcal{C} denoted as C_1, \dots, C_n . The clients communicate with the server by sending messages over reliable channels directly to the server, forming an

asynchronous distributed system. Any number of the clients may be *crash-faulty* while server S is *Byzantine-faulty* [PSL80], i.e., it may deviate arbitrarily from its algorithm [JCT98].

The two distributed protocols introduced in this chapter implement a shared object with functionality of a *shared memory* (cf. Section 2.1.3) providing *operations* at the clients' interface. To represent *abortable* operations, here, in addition to the given invocation events, there are two types of response events: ABORT and OK. In history σ of the execution of a distributed protocol, we call operation op *complete*, if there exists a matching response event to the invocation event of op , else op is denoted as *incomplete*. An operation is *successful*, iff it is complete and the response event is an OK event. An operation is *aborted*, if it is complete and the response event is an ABORT event. Operation op *precedes* operation op' iff op is complete before the invocation event of op' . If op precedes op' we denote op and op' as *sequential* operations. Else, if neither operation precedes the other, then op and op' are said to be *concurrent*.

A shared memory provides READ and WRITE operations, such that $\forall i = 1, \dots, n$, client C_i provides both operations $\text{WRITE}(i, v)$ and $\text{READ}(i)$ while all other clients only provide operation $\text{READ}(i)$ (cf. Section 2.1.3). A shared memory can also be regarded as a collection of atomic registers, where each client may write to a dedicated register but may only read from all other registers. We assume that each client interacts *sequentially* with the shared memory, i.e., a client invokes a new operation only after the previous operation has completed.

Further we assume that clients have access to a digital signature scheme used by each client to *sign* its messages such that any other client can determine the authenticity of a message by *verifying* the corresponding signature. Further, the Byzantine server S is not able to forge the signatures.

The consistency condition for the shared memory is defined in terms of the history σ of an execution of the distributed protocol. To ease the definition of consistency conditions and the reasoning about correctness, we define two transformations to derive simpler histories from more complicated ones, while maintaining plausibility of execution. Intuitively, the transformations remove all operations from a history that do not take effect.

Definition 3.1. An operation op of client *takes effect* if and only if

1. op is *successful* OR
2. op is a WRITE operation *and* there exists a READ operation that returns the value written by op .

We now define the two transformations CRASHCOMPLETE and ABORTCOMPLETE.

Definition 3.2. The transformations `CRASHCOMPLETE` and `ABORTCOMPLETE` take a history σ as input and return a sequence of events σ' as output.

- `CRASHCOMPLETE`: We define σ' returned by `CRASHCOMPLETE` by construction: At first we add all events from σ to σ' . Then, we remove the invocation events of *incomplete* operations that did not take effect from σ' . Next, we add a matching OK event to each remaining *incomplete* operation in σ' .
- `ABORTCOMPLETE`: We define σ' returned by `ABORTCOMPLETE` by construction: At first we add all events from σ to σ' . Then, we remove all events of aborted operations in σ' that did not take effect. Next, we replace all remaining ABORT events in σ with matching OK events.

Transformation `CRASHCOMPLETE` removes incomplete operations that did not take effect from σ . This is reasonable as such events do not influence the execution. Instead of removing them, such events could also be moved to the end of sequence σ . The same argument applies to aborted operations that do not take effect which are removed by transformation `ABORTCOMPLETE`. By first applying transformation `CRASHCOMPLETE` and then transformation `ABORTCOMPLETE` to sequence σ , we have transformed σ into a sequence of events containing only *successful* operations. On the transformed sequence we give the definitions of *fork-linearizability* taken from recent work of Cachin *et al.* [CSS07].

Definition 3.3 (Global Fork-Linearizability). A history σ of an execution of a distributed protocol is called *fork linearizable* with respect to a functionality F if and only if there exists a sequential permutation π of σ such that:

1. π preserves the real-time order of σ ; and
2. for each client C_i there exists a subsequence π_i of π such that:
 - a) events in π occurring at client C_i are contained in π_i ; and
 - b) the operations of π_i satisfy the sequential specification of F ; and
 - c) for every $op \in \pi_i \cap \pi_j$, the sequence of events that precede op in π_i is the same as the sequence of events that precede op in π_j .

Using a definition of fork-linearizability that is different from the one given in Definition 2.3 in Section 2.2.2 simplifies the correctness proofs of the protocols `LINEAR` and `CONCUR`. The notions of fork-linearizability and global fork-linearizability have shown to be equivalent [CSS07].

3.4 Preliminaries on the Protocols

In the next sections we present two lock-free protocols LINEAR and CONCUR that emulate a fork-linearizable shared memory on a Byzantine server — constituting contributions (C1) and (C2), respectively, of this thesis. The LINEAR protocol is based on *vectors* of timestamps (described later in section 3.5) resulting in a communication complexity of $\mathcal{O}(n)$. The LINEAR protocol serializes all operations, and therefore it aborts concurrent operations even if they are applied to distinct registers of the shared memory. The CONCUR protocol (introduced later in section 3.6) allows for concurrent operations if they are applied to distinct registers and only operations on the same register are serialized. To achieve this, timestamp *matrices* are used leading to a communication complexity of $\mathcal{O}(n^2)$.

Protocol Properties

As mentioned above, the LINEAR and CONCUR protocol emulate the functionality of a shared memory with fork-linearizability among a collection of clients and a (possibly) Byzantine server S (cf. Definition 2.4 in Section 2.2.2). The LINEAR (CONCUR) protocol consists of two algorithms, run by the clients and the server respectively. If the server is faulty, it may refuse to respond to client requests or return (detectably) corrupted data. A malicious server may also mount a forking attack and partition clients. However, if the server behaves correctly, we require that the emulation does not block and clients are not forked.

To formalize the liveness properties of the LINEAR and CONCUR protocol, we redefine the notion of *sequential* and *concurrent* operations under step contention [AGK05] when the server is correct. We say that two operations op and op' are *sequential under step contention* if op' does not perform steps at the server S after op performed its first step and before op performed its last step at server S . Otherwise, op and op' are *concurrent under step contention*.

The *Byzantine emulation* of a shared memory with fork-linearizability implemented by the LINEAR and CONCUR protocol satisfies the following two liveness properties *Nontriviality* and *Termination*:

Nontriviality: When the server is correct, in an execution of the LINEAR (resp. CONCUR) protocol every operation that returns *abort* is concurrent under step contention with another operation (resp. with another operation on the same register).

Termination: When the server is correct and σ is the history of an execution of the LINEAR or CONCUR protocol, then after applying transformation CRASHCOMPLETE to σ , every operation in σ is complete.

3.5 (C1): The LINEAR Protocol

The LINEAR protocol is based on two main ideas. The first idea is that when two or more operations access the registers of the shared memory concurrently, all but one are aborted. In the protocol, operations need two rounds of communication with the server, and an operation op is aborted if a first round message of another operation arrives at the server between the points in time when the first round message and the second round message of op is received by the server. Hence, among the concurrent operations, the LINEAR protocol never aborts the “newest” operation. This scheme ensures that an incomplete operation of a crashed client does not interfere with other operations. Note that by employing this strategy of aborting, successful operations execute in isolation and therefore accesses to the shared memory are serialized.

As a second idea, the LINEAR protocol assigns vector timestamps to operations such that a partial order \leq on operations can be defined based on these timestamp vectors. The basic principle is that a client reads the most recent timestamp vector from the server during the first round, increments its own entry and writes the updated timestamp vector back to the server. Since successful operations run in isolation, the corresponding timestamp vectors are totally ordered, as no two successful operations read the same timestamp vector during the first round. Clearly, a Byzantine server may fork two clients, but then there are operations op and op' of these two clients with incomparable timestamp vectors. By the requirement of fork-linearizability, these two clients must not see any later updates of each other. For this purpose, the protocol ensures that the two clients remain forked by preventing any client from committing an operation op'' which is both greater (with respect to partial order leq) than op and op' .

3.5.1 Description of the LINEAR Protocol

The shared memory implemented by the LINEAR protocol provides n SWMR registers $X[1], \dots, X[n]$ such that client C_i may write a value only to register $X[i]$ and may read from any register. The detailed pseudo-code of the LINEAR protocol appears in Algorithms 3.1 and 3.3 giving the algorithm of a client C_i , and in Algorithm 3.2 describing the algorithm executed by server S .

A client performs two rounds of communication with the server S for both READ and WRITE operations (see Algorithm 3.1). This is implemented by calling procedure `RW_OPERATION` (Algorithm 3.3) with type `READ` or `WRITE` respectively. When executing a request to operation `RW_OPERATION`, the client sends a `SUBMIT` message to the server S announcing a read or write operation and waits for a matching response. The server S responds with a `SUBMIT_R` message containing information on the current state of the server and the value to be read. In the

Algorithm 3.1: Read/Write Operation of Client i

```

3.1.1 READ( $j$ ) do
3.1.2   RW_OPERATION(READ,  $\perp$ ,  $j$ )
3.1.3   if abort then return ABORT
3.1.4   return retval

3.1.5 WRITE( $i, v$ ) do
3.1.6   RW_OPERATION(WRITE,  $v, i$ )
3.1.7   if abort then return ABORT
3.1.8   return OK

```

Algorithm 3.2: LINEAR Protocol, Algorithm of Server S

Variables:

```

3.2.1 Pnd set of operation ids                /* pend.  ops                */
3.2.2 Abt set of operation ids                /* pending ops to be aborted */
3.2.3 upon receiving message  $\langle \text{SUBMIT}, id \rangle$  from client  $i$  do
3.2.4    $Abt \leftarrow Pnd$ 
3.2.5    $Pnd \leftarrow Pnd \cup \{id\}$ 
3.2.6   send  $\langle \text{SUBMIT\_R}, X[id.reg], lso \rangle$  to client  $i$ 
3.2.7 upon receiving message  $\langle \text{COMMIT}, op \rangle$  from client  $i$  do
3.2.8    $Pnd \leftarrow Pnd \setminus \{op.id\}$ 

3.2.9   if  $op.id \in Abt$  then
3.2.10    send  $\langle \text{COMMIT\_R}, \text{ABORT} \rangle$  to client  $i$ 
3.2.11   else
3.2.12     $X[i] \leftarrow op$ 
3.2.13     $lso \leftarrow op$ 
3.2.14    send  $\langle \text{COMMIT\_R}, \text{OK} \rangle$  to client  $i$ 

```

second communication round, the client sends a COMMIT message to the server and waits for a COMMIT_R message to complete the operation. The COMMIT_R message is either of type OK or ABORT indicating to the client the outcome of the operation.

Variables and Data Structures Each operation op has a timestamp vector of size n assigned to it during the protocol. The timestamp vector is part of the operation data structure and is denoted as $op.tsv$. The timestamp vector is used to define a partial order \leq on operations. For two operations op and op' we say that $op \leq op'$ iff $op.tsv[i] \leq op'.tsv[i]$ for all $i = 1 \dots n$. Operations of the LINEAR protocol have the data structure of a 4-tuple with entries id , $value$, tsv and sig , where sig is a signature on the operation by the client, tsv is the timestamp vector, $value$ is

the value to be written by the operation. Note that for simplicity of presentation, a READ operation rewrites the value of the client's last successful WRITE. The entry id is a 4-tuple $\langle client_id, op_cnt, type, reg \rangle$ itself, where $client_id$ equals i for C_i , op_cnt is a local timestamp of the client which is incremented during every operation, $type$ indicates whether the operation is a READ or a WRITE, and reg determines the index of the register the client intends to read from. For WRITE operations of client C_i , reg is always i .

The server S maintains the n registers in a vector $X[1..n]$, where each $X[i]$ stores the last successful operation of C_i . Further, the server maintains a copy of the latest successful operation in variable lso .

Initially, variables op_cnt , ts_{suc} , and all entries of vector $tsv_{comp}[1..n]$ at any client are assumed to be 0. At initialization of the server, variable lso and all registers $X[i]$ store the initial operation op_0 defined as $\langle \langle i, 0, \text{WRITE}, i \rangle, \perp, [0..0], sig_0 \rangle$ where sig_0 is a valid signature on operation op_0 . Sets Pnd and Abt are empty.

Processing Operations When client C_i invokes a new operation op on register $X[r]$, it increments its local timestamp op_cnt , sets the entries of $op.id$ to the operation type and register r , and sends $op.id$ in a SUBMIT message to the server (lines 3.3.2–3.3.5). The server labels the received operation op as *pending*. If the server receives the SUBMIT message of another operation before the COMMIT message of op , then op is aborted. In reply to the received SUBMIT message, the server responds with a SUBMIT_R message containing the last successful operation lso , and the last successful operation x_op applied to register $X[r]$ (lines 3.2.4–3.2.6).

After receiving operations lso and x_op from the server, client C_i performs a number of consistency checks (lines 3.3.7–3.3.10). If any of the checks fails, which implies that the server is misbehaving, the client halts. In the first check, C_i verifies the signatures of lso and x_op . The next check is needed to determine a consistent timestamp vector for operation op . The goal is to obtain a timestamp vector for op which is greater than both lso 's timestamp vector and that of C_i 's last *completed* operation. The timestamp vector of the latter is stored in tsv_{comp} at C_i . The client checks that all but the i th entry in $lso.tsv$ are greater or equal than the corresponding entries in tsv_{comp} . C_i 's entry $lso.tsv[i]$ must equal the timestamp of the last *successful* operation stored in ts_{suc} . Checks three and four are needed only by READ: C_i checks that x_op is indeed the content of register $X[r]$. The last check verifies that lso is at least as large as x_op and that $lso.tsv[r]$ equals $x_op.tsv[r]$.

If all checks are passed, C_i increments its own entry $lso.tsv[i]$ and $lso.tsv$ becomes the timestamp vector of op . Then, C_i signs $op.id$, the write value and the timestamp vector $op.tsv$, and sends op in a COMMIT message to the server (lines 3.3.11–3.3.15). The server, removes $op.id$ from the set of pending operations and

checks if it has to be aborted. As mentioned earlier, in this case, a SUBMIT message of another operation was received before the COMMIT of op and the server replies with ABORT (lines 3.2.8–3.2.10). Else, op is stored in $X[i]$ and also in lso as the last successful operation and the server replies with OK (lines 3.2.12–3.2.14).

When client C_i receives the COMMIT_R message for operation op , op is completed and thus tsv_{comp} is updated with $op.tsv$. If op is successful, then additionally ts_{suc} becomes the i th entry of $op.tsv$. If op is a READ, then the value of x_{op} is returned (lines 3.3.17–3.3.24).

Algorithm 3.3: LINEAR Protocol, Algorithm of Client i

Variables:

sig signature /* signature */
 $abort$ boolean /* flags if operation is aborted */
 $value_{suc}, retval$ value /* last successful write + return value */
 op_cnt, ts_{suc} integer /* op counter + ts last successful op */
 op, x_{op}, lso operation with fields
 $id = \langle client_id, op_cnt, type, reg \rangle, value, tsv, sig$ /* operation */
 $tsv_{comp}[1..n]$ vector of integers /* ts vector of last comp. op */

```

3.3.1 RW_OPERATION(TYPE, value, r)
3.3.2   abort ← false
3.3.3   op_cnt ← op_cnt + 1
3.3.4   op.id ← (i, op_cnt, TYPE, r)
3.3.5   send ⟨SUBMIT, op.id⟩ to server
3.3.6   wait for message ⟨SUBMIT_R, x_op, lso⟩
3.3.7   if not verify(lso.sig) ∧ verify(x_op.sig) then halt
3.3.8   if not ∀k ≠ i : tsv_comp[k] ≤ lso.tsv[k] ∧ ts_suc = lso.tsv[i] then halt
3.3.9   if not x_op.id.client_id = r then halt
3.3.10  if not x_op ≤ lso ∧ lso.tsv[r] = x_op.tsv[r] then halt
3.3.11  op.tsv ← lso.tsv
3.3.12  op.tsv[i] ← op_cnt
3.3.13  if TYPE = WRITE then op.value ← value
3.3.14  op.sig ← sign(op.id || op.value || op.tsv)
3.3.15  send ⟨COMMIT, op⟩ to server
3.3.16  wait for message ⟨COMMIT_R, ret_type⟩
3.3.17  tsv_comp ← op.tsv
3.3.18  if ret_type = ABORT then
3.3.19     op.value ← value_suc
3.3.20     abort ← true
3.3.21  else
3.3.22     ts_suc ← op_cnt
3.3.23     value_suc ← op.value
3.3.24     if TYPE = READ then retval ← x_op.value

```

3.5.2 Correctness Arguments

Instead of returning the most recent value that has been written to register $X[j]$ by a WRITE operation op_w , a Byzantine server may return an old value written by operation op'_w . Let C_i be the client whose READ operation op_r reads the stale value written by op'_w . Note that the Byzantine server also returns a stale version of lso to C_i . Let us assume that all checks in Algorithm 3.3 are passed, thus C_i is unaware of the malicious behavior of the server. Note, that the j th entry in the timestamp vector of op'_w is smaller than the corresponding entry in op_w , as both are operations of client C_j that increases the j th entry with every operation. As the check in line 3.3.10 is passed, the j th entry in op_r 's timestamp vector is also smaller than the one of op_w . As C_i increments the i th entry in the timestamp vector during op_r but not the j th entry, op_r and op_w are incomparable (i.e., neither $op_r \leq op_w$ nor $op_w \leq op_r$). We argue that in this situation, no client commits an operation which is greater than both op_w and op_r . As no client other than C_i increments the i th entry in a timestamp vector, all operation of other clients that “see” op_w have a timestamp vector whose i th entry is smaller than $op_r.tsv[i]$ and whose j th entry is larger than $op_r.tsv[j]$. Thus, such operations are also incomparable with op_r and do not join op_w and op_r . When client C_i “sees” such an operation incomparable to op_r as the latest successful operation lso , the check in line 3.3.8 is not passed because the i th entry of lso is smaller than the timestamp of C_i 's last successful operation. Hence, C_i stops the execution. Analogously, the same arguments can be applied for client C_j and operation op_w .

As all checks are passed when the server behaves correctly, it is not difficult to see that with a correct server, all operations invoked by correct clients complete. Also with a correct server, operations are only aborted in the specified situations. A detailed correctness proof is given in the next section closing the presentation of the LINEAR protocol in this thesis.

3.5.3 LINEAR Protocol Proof

This section gives the proof of correctness that the LINEAR protocol constitutes an abortable, Byzantine emulation of a shared memory with fork-linearizability. Before beginning the proof, the notion of an *operation*, as used in the LINEAR protocol in Algorithm 3.3 and 3.2, is formally defined.

Definition 3.4 (LINEAR Operation). An *operation identifier* (*operation id*) is a 4-tuple $\langle client_id, op_cnt, type, reg \rangle$, where $client_id$, op_cnt , and reg are integers and where $type$ is element of the set $\{\text{READ}, \text{WRITE}\}$.

An *operation* is a 4-tuple $\langle id, value, tsv, sig \rangle$, where id is an operation identifier, $value$ is a value from set \mathcal{V} , tsv is a vector of size n of integers, and sig is a signature.

We define a partial order \leq on timestamp vectors and on operations. Note, that we regard only such operations op after the corresponding timestamp vector entry $op.tsv$ has been assigned in line 3.3.12.

Definition 3.5 (Order Relation). For two timestamp vectors tsv and tsv' holds $tsv \leq tsv'$ if and only if

$$\forall i : tsv[i] \leq tsv'[i].$$

It holds $tsv = tsv'$ if and only if tsv and tsv' are the same timestamp vectors.

For two operations op and op' holds $op \leq op'$ if and only if

$$op.tsv \leq op'.tsv.$$

It holds $op = op'$ if and only if op and op' are the same operations.

It is easy to see that \leq relation on operations (timestamp vectors) is *transitive*. As relation \leq is a partial order on operations (timestamp vectors), we define a notion of when two operation (timestamp vectors) cannot be ordered by \leq .

Definition 3.6 (Comparable). For two timestamp vectors tsv and tsv' holds tsv and tsv' are *comparable* if and only if

$$tsv \leq tsv' \vee tsv' \leq tsv.$$

Otherwise, they are *incomparable*.

For two operations op and op' holds op and op' are *comparable* if and only if

$$op.tsv \text{ and } op'.tsv \text{ are comparable.}$$

Otherwise, they are *incomparable*. We also call two incomparable operations *forked*.

The next Lemma shows that \leq relation on LINEAR operations does not violate the real-time order of operations.

Lemma 3.7. If $op \leq op'$ then op' does not precede op .

Proof. Let op and op' be two operations of client C_i and C_j and let us assume by contradiction that op' precedes op and $op \leq op'$. During op , client C_i updates the i th entry in the timestamp vector (line 3.3.12). As op' precedes op and as the server cannot forge signatures (line 3.3.14), at the point in time when C_j received the SUBMIT_R message during op' , there exists no operation op'' such that $op''.tsv[i] \geq op.tsv[i]$. Thus, we have that $op.tsv[i] > op'.tsv[i]$. However, this contradicts the assumption that $op \leq op'$. \square

The following two Lemmas show that operations which causally influence each other are ordered by \leq such that the causal order is respected. The operations of one client causally influence each other (Lemma 3.8) as well as a WRITE operation and an operation which reads the written value (Lemma 3.9).

Lemma 3.8. All operations of the same client are totally ordered by \leq relation on operations.

Proof. We show that operation op of client C_i is greater than its previous completed operation op_{comp} . Note, that by line 3.3.17 $op_{comp}.tsv = tsv_{comp}$. Let l be operation lso as received in the SUBMIT_R message by C_i during operation op (line 3.3.6). To pass the check in line 3.3.8, l must be greater or equal in all entries $\neq i$ of the timestamp vector than op_{comp} . By line 3.3.11 we have that $op.tsv[k] \geq op_{comp}.tsv[k]$ for all $k \neq i$. In line 3.3.12 the i th entry of the timestamp vector is updated by a larger entry, as op_cnt is incremented with every invoked operation of C_i (line 3.3.3), and we get that $op.tsv > op_{comp}.tsv$, implying that $op > op_{comp}$. By induction on C_i 's operations, it follows that op is greater than any operation of C_i that precedes op . \square

Lemma 3.9. If op_r is a READ(j) operation of client C_i that returns $op_w.value$ from the shared memory, then $op_w < op_r$.

Proof. To pass the check in line 3.3.10, it must be that $op_w \leq lso$ and by lines 3.3.11 and 3.3.12 it holds that $lso < op_r$. Thus, if op_r returns $op_w.value$ it must be that $op_w < op_r$. \square

The next Lemma proves the main result that the LINEAR protocol satisfies fork-linearizability according to Definition 3.3 on page 38.

Lemma 3.10. The LINEAR protocol described in Algorithm 3.1, 3.2, and 3.3 and emulates a shared memory on a Byzantine server with fork-linearizability satisfying properties *nontriviality* and *termination* (cf. Section 3.4).

Proof. Let σ be the history of any execution of the LINEAR protocol. At first, we apply transformation CRASHCOMPLETE to σ (Definition 3.2). We construct a sequential execution π by totally ordering all events in σ . To achieve this, we order the events in σ by the following rules:

1. Sort the operations in σ by \leq relation on operations.
2. Sort any yet unsorted operations by the real-time order of their completion event in σ .

We construct the subsequences π_i (for $i = 1, \dots, n$) as required by the definition of fork-linearizability. We include in π_i all operations of client C_i . Then, for all $op \in \pi_i$ we include into π_i all operations op' in σ such that $op' \leq op$.

By Lemma 3.7, the following claim follows directly:

Claim 10.1 Let op and op' be two operations and op precedes op' in σ . Then, op precedes op' in π .

Claim 10.2 Let op_r be a completed $\text{READ}(j)$ operation of client C_i in some sequence π_k that returns $op_w.\text{value} \neq \perp$ from the shared memory. Then:

1. Operation op_w^1 is in π_k , and
2. there is no $\text{WRITE}(j, *)$ operation by client C_j subsequent to op_w in π_k that takes effect and completes before op_r is invoked.

By Lemma 3.9 holds that $op_r > op_w$. Hence, op_w is included in π_k by construction and the first statement follows directly.

To prove the second statement, let us assume for contradiction that such a WRITE operation op'_w of client C_j exists in π_k which is invoked after op_w completes and which completes before op_r is invoked. Hence, op_w, op'_w, op_r appear in that order as three sequential operations in σ .

We first show that op'_w and op_r are incomparable: Let l be operation lso as seen by op_r (line 3.3.6). Since $i \neq j$ the j th entry of l is not changed during op_r and thus $op_r.\text{tsv}[j] = l.\text{tsv}[j]$. To pass the check in line 3.3.10, we also have $op_w.\text{tsv}[j] = l.\text{tsv}[j]$. Moreover, since op_w and op'_w are both operations of the same client and op_w precedes op'_w we have by line 3.3.3 $op'_w.\text{tsv}[j] > op_w.\text{tsv}[j]$ and thus

$$op_r.\text{tsv}[j] = op_w.\text{tsv}[j] < op'_w.\text{tsv}[j]. \quad (3.1)$$

By line 3.3.12 we know that client C_i is the only client that increments the i th entry in any valid timestamp vector. Thus, the i th entry of any timestamp vector before the invocation of op_r is less than $op_r.\text{tsv}[i]$. This applies also to op'_w and we have

$$op_r.\text{tsv}[i] > op'_w.\text{tsv}[i]. \quad (3.2)$$

According to Definition 3.6, equations (3.1) and (3.2) imply that operations op'_w and op_r are incomparable. In the following we distinguish two cases:

Case 1 Operation op'_w is successful. Both operations op_r and op'_w are contained in π_k . Thus, by construction of π_k there must exist operations op, op' of client C_k such that $op_r \leq op$ and $op'_w \leq op'$. As all operations of client C_k

¹Note that op_w is a $\text{WRITE}(op_w.\text{value}, j)$ operation of client C_j .

are totally ordered, either $op \leq op'$ or $op > op'$. Let us assume w.l.o.g. that $op \leq op'$ and thus, $op' \geq op_r$ and $op' \geq op'_w$.

Since op_r is in π_k there must exist a sequence of operations $S_{op} : op_r, op_{s1}, \dots, op_{s(z-1)}, op'$ such that $op_r = op_{s0}$, $op' = op_{sz}$, and $\forall i = 0, \dots, z$ holds op_{si} is *lso* as seen by $op_{s(i+1)}$. Thus, it holds $op_r < op_{s1} < \dots < op_{s(z-1)} < op'$ and each timestamp vectors $op_{si}.tsv$ and $op_{s(i+1)}.tsv$ differ only in one entry.

By equation (3.1) we know that $op_r.tsv[j] < op'_w.tsv[j]$. As no other client than C_j increments the j th entry of a timestamp vector, no operation of C_j may be in sequence S_{op} . Otherwise, as op'_w is successful, $lso.tsv[j] < ts_{suc}$ and the check in line 3.3.8 is not passed.

Case 2 Operation op'_w is not successful. To ensure that op'_w is in π_k there must be a successful $READ(j)$ operation op'_r of client C_l that is in π_k and reads $op'_w.value$. Since operations op_r and op'_r are both contained in π_k by assumption, there exist operations op, op' of client C_k such that $op_r \leq op$ and $op'_r \leq op'$. As all operations of client C_k are totally ordered, either $op \leq op'$ or $op > op'$. Let us assume w.l.o.g. that $op \leq op'$ and thus, $op' \geq op_r$ and $op' \geq op'_r$. We may now deduce the following statements:

- (i) As op_r reads $op_w.value$ and $l \neq i$, we have that $op_r.tsv[l] = op_w.tsv[l]$. As op_w precedes op'_w and op'_r reads $op'_w.value$, we conclude that $op'_r.tsv[l] > op'_w.tsv[l] \geq op_w.tsv[l] = op_r.tsv[l]$.
- (ii) By equation (3.2) we know that $op'_w.tsv[i] < op_r.tsv[i]$. As op'_r reads $op'_w.value$ and client C_l does not increment the i th entry ($i \neq l$), we get $op'_r.tsv[i] < op_r.tsv[i]$.

Since op_r is in π_k there must exist a sequence of operations $S_{op} : op_r, op_{s1}, \dots, op_{s(z-1)}, op'$ such that $op_r = op_{s0}$, $op' = op_{sz}$, and $\forall i = 0, \dots, z$ holds op_{si} is *lso* as seen by $op_{s(i+1)}$. Thus, it holds $op_r < op_{s1} < \dots < op_{s(z-1)} < op'$ and each timestamp vectors $op_{si}.tsv$ and $op_{s(i+1)}.tsv$ differ only in one entry.

By construction of π_k we know that $op' \geq op'_r$ and by (i) we have $op'_r.tsv[l] > op_r.tsv[l]$. Hence, for $tsv_{comp}[l]$ at the beginning of op' holds $tsv_{comp}[l] > op_r.tsv[l]$. This means that if sequence S_{op} did not contain any operation of C_l , op' would not accept $op_{s(z-1)}$ as *lso* because $op_{s(z-1)}.tsv[l] < tsv_{comp}[l]$. Hence there must exist some operation op_l of client C_l in sequence S_{op} . Let op_l be the first such operation. We distinguish the following two cases:

- (a) op_l precedes op'_r at client C_l . As op_l is contained in sequence S_{op} we know by construction that $op_l.tsv[i] \geq op_r.tsv[i]$. This implies that at client C_l after completing operation op_l it holds that $tsv_{comp}[i] \geq op_r.tsv[i]$. During op'_r only the l th entry is changed, i.e., $lso.tsv[i] = op'_r.tsv[i]$. Hence, as op_l precedes op'_r , during op'_r it holds $tsv_{comp}[i] >$

$lso.tsv[i]$ (by statement (ii)) and check in line 3.3.8 is not passed, a contradiction.

- (b) op'_r precedes op_l at client C_l . Since op'_r is a successful operation, during op_l holds that $ts_{suc} \geq op'_r.tsv[l]$. By statement (i) it follows that $ts_{suc} > op_r.tsv[l]$. As op_l is the first operation of C_l in sequence S_{op} , lso as seen by op_l does not pass the check in line 3.3.8, a contradiction.

Hence, there exists no WRITE operation op'_w of client C_j in π_k which is invoked after op_w completes and which completes before op_r is invoked and the second statement of Claim 10.2 is also true and we are done. \square

Lemma 3.10 closes the part of the correctness proof of the LINEAR protocol regarding the safety properties. We now continue by proving that the LINEAR protocol also satisfies the specified liveness properties. The next lemma proves that as long as server S behaves correctly, all operations eventually complete (by either returning ABORT or OK).

Lemma 3.11. If the server is correct then no operation in Algorithm 3.3 blocks.

Proof. We have to show that no operation blocks in lines 3.3.7 – 3.3.10.

- $\text{verify}(lso.sig) \wedge \text{verify}(x_op.sig)$ is TRUE: As clients are non-malicious, all signatures are correct. Thus, the protocol does not block in line 3.3.7.
- Assume by contradiction that $\exists k : tsv_{comp}[k] > lso.tsv[k]$: Let op_{comp} be the last completed operation that updated tsv_{comp} in line 3.3.17. As the server is correct it returns only successful operations, so particularly operation lso is successful. By line 3.2.12 and 3.2.13 whenever some operation successfully completes, lso is also updated. Thus, as lso is monotonically increasing (Lemma 3.7 and Lemma 3.10), $\forall k : op_{comp}.tsv[k] \not> lso.tsv[k]$, which is a contradiction.

Assume by contradiction that $ts_{suc} \neq lso.tsv[i]$: Let op_{suc} be the last successful operation that updated ts_{suc} . By the reasoning above, whenever some register at the server is updated, lso is also updated. This implies that $lso.tsv[i] \not> op_{suc}.tsv[i]$. Therefore, it must be that $lso.tsv[i] > op_{suc}.tsv[i]$. Entry $lso.tsv[i]$ is only updated when an operation of C_i later than op_{suc} successfully completes. However, in this case ts_{suc} is also updated to the same value and we have $lso.tsv[i] = ts_{suc}$, a contradiction.

Thus, the protocol does not block in line 3.3.8.

- $x_op.id.client = r$ is TRUE: As the server is correct it returns x_op from the correct register. Thus the protocol does not block in line 3.3.9.

- Assume by contradiction that $x_op \not\leq lso$: As the server is correct it returns only successful operations. Thus, both x_op and lso are successful. This implies that x_op and lso can be ordered by \leq . By line 3.2.12 and 3.2.13 whenever some register at the server is updated, lso is also updated. Thus, as lso is monotonically increasing, $x_op \not\leq lso$. Therefore, $x_op \leq lso$ which is a contradiction.

Assume by contradiction that $lso.tsv[r] \neq x_op.tsv[r]$: By the item above, whenever some register at the server is updated, lso is also updated. This implies that $lso.tsv[r] \not\leq x_op.tsv[r]$. Therefore, it must be that $lso.tsv[r] > x_op.tsv[r]$. Entry $lso.tsv[r]$ is only updated when an operation of client C_r later than x_op successfully completes. However, in this case $x_op.tsv[r]$ is also updated to the same value and we have $lso.tsv[r] = x_op.tsv[r]$, a contradiction.

Thus, the protocol does not block in line 3.3.10

Hence, no operation blocks in lines 3.3.7 – 3.3.10 and we are done. \square

The next lemma shows that the LINEAR protocol satisfies the *nontriviality* property when server S is correct. This property rules out trivial implementations where every operation is aborted.

Lemma 3.12. With a correct server, an operation op of a client aborts only if the server receives a SUBMIT message from another client after the SUBMIT and before the COMMIT message corresponding to op .

Proof. If op aborts then the server has received the COMMIT message corresponding to op . As the server is correct and no SUBMIT message of another operation is received after the SUBMIT and before the COMMIT message of op , $op.id \notin Abt$. By line 3.2.12 and 3.2.13, op is not aborted. \square

Finally, the following theorem proofs the correctness of the LINEAR protocol and closes Section 3.5.

Theorem 3.13. The LINEAR protocol is a non-trivial, abortable, Byzantine emulation of a shared memory with fork-linearizability (cf. Section 3.4).

Proof. By Lemma 3.10, the safety properties of fork-linearizability are satisfied, by Lemma 3.11, the protocol does not block, and by Lemma 3.12, no operation running in isolation is aborted when the server is correct. \square

3.6 (C2): The CONCUR Protocol

The CONCUR protocol differs from the LINEAR protocol in the way how concurrent access to the server is handled. In contrast to the LINEAR protocol, in the CONCUR protocol concurrent operations that access different registers of the shared memory are not aborted. Intuitively, the same aborting scheme as in the LINEAR protocol is used in the CONCUR protocol on a per register basis in order to serialize all accesses to the same register of the shared memory. This means, that a correct server aborts operation op accessing register i if and only if a SUBMIT message of another operation accessing register i is received while op is pending.

To deal with concurrent operations, in the CONCUR protocol, instead of one timestamp vector, each operation is assigned n timestamp vectors, each corresponding to one register of the shared memory. Such n timestamp vectors form the *timestamp matrix* of an operation. The basic idea is that when a client accesses register j then the client updates its own entry in the j th timestamp vector of the timestamp matrix. It is important to note that even with a correct server, the CONCUR protocol allows that two clients with concurrent operations may read the same timestamp matrix from the server and update different timestamp vectors such that the corresponding operations become incomparable. However, the CONCUR protocol ensures that (1) operations of the same client are totally ordered by \leq and (2) operations accessing the same register at the server are totally ordered by \leq . This is sufficient to show that for any operation op , all operations op causally depends on, are ordered before op by \leq . Further, the CONCUR protocol ensures that two forked operations — i.e., for some i , the i th timestamp vectors in the timestamp matrices of the two operations are incomparable — will never be rejoined by another operation.

3.6.1 Description of the CONCUR Protocol

The CONCUR protocol has the same message pattern as the LINEAR protocol and provides the same interface to the clients (Algorithm 3.1). The CONCUR protocol uses a different implementation of procedure RW_OPERATION as described in Algorithm 3.4 (code of the server), and Algorithm 3.5 (code of the clients). As the CONCUR protocol follows the structure of the implementation of the LINEAR protocol, in the following we highlight only the differences between the two protocols. The *operation* data structure differs from the LINEAR protocol only to the fact that the timestamp vector tsv is replaced by a timestamp matrix tsm .

When client C_i invokes a new operation op on register r of the shared memory, it generates a new operation id which it sends to the server in a SUBMIT message (lines 3.5.2–3.5.5). One difference is that C_i maintains a *separate* operation counter for each register $op_cnt[1..n]$. The server replies with operations lso and x_op

Algorithm 3.4: CONCUR Protocol, Algorithm of Server S

Variables:

```

3.4.1  $Pnd[1..n]$  array of set of operation ids          /* pending ops          */
3.4.2  $Abrt[1..n]$  array of set of operation ids        /* pending ops for abort */
3.4.3 upon receiving message  $\langle \text{SUBMIT}, id \rangle$  from client  $i$  do
3.4.4    $Abrt[id.reg] \leftarrow Pnd[id.reg]$ 
3.4.5    $Pnd[id.reg] \leftarrow Pnd[id.reg] \cup \{id\}$ 
3.4.6   send  $\langle \text{SUBMIT\_R}, X[id.reg], lso[id.reg] \rangle$  to client  $i$ 
3.4.7 upon receiving message  $\langle \text{COMMIT}, op \rangle$  from client  $i$  do
3.4.8    $Pnd[op.id.reg] \leftarrow Pnd[op.id.reg] \setminus \{op.id\}$ 
3.4.9   if  $op.id \in Abrt[op.id.reg]$  then
3.4.10    send  $\langle \text{COMMIT\_R}, \text{ABORT} \rangle$  to client  $i$ 
3.4.11   else
3.4.12     $X[i] \leftarrow op$ 
3.4.13     $lso[op.id.reg] \leftarrow op$ 
3.4.14    send  $\langle \text{COMMIT\_R}, \text{OK} \rangle$  to client  $i$ 

```

contained in a SUBMIT_R message. Here, x_op is the last successful operation stored in register r , and lso is the last successful operation that accessed register r . Note, that lso may not be stored in register r — e.g., x_op might be a $\text{WRITE}(i, *)$ operation while lso might be a $\text{READ}(i)$ operation of client C_j which is not stored in register r . The server maintains information on pending operations for each register separately (lines 3.4.4–3.4.6).

The first and the third consistency check are identical to the LINEAR protocol. The second check on operations lso and x_op performed by the client corresponds to the second check in the LINEAR protocol. As CONCUR operations hold a timestamp matrix, the check is performed on the r th timestamp vectors of the timestamp matrices of lso and x_op . The goal is to obtain a timestamp matrix for op which is greater than the last completed operation of C_i and the last successful operation accessing register r , stored in lso . Like in the LINEAR protocol, the last check ensures that lso is greater than x_op and, unlike LINEAR, that the r th entries in the r th timestamp vector of the timestamp matrices of lso and x_op are equal. This particular entry is the one which has been updated during x_op (lines 3.5.7–3.5.10).

To determine the timestamp matrix for op , client C_i selects the r th timestamp vector from lso as r th timestamp vector of op and for all other indices it takes the maximum timestamp vector from lso and C_i 's last completed operation. Finally, client C_i increments its own entry in the r th timestamp vector using $op_cnt[r]$ (lines 3.5.12–3.5.15). The remainder of the protocol is analogous to the LINEAR protocol.

Algorithm 3.5: CONCUR Protocol, Algorithm of Client i

Variables:

sig	signature	/* signature */
$abort$	boolean	/* flags if operation is aborted */
$value_{suc}, retval$	value	/* last successful write + return value */
$op_cnt[1..n], ts_{suc}[1..n]$	vector of integers	/* vectors */
op, x_op, lso	operation with fields	
$id = \langle client_id, op_cnt, type, reg \rangle$		/* operation */
$tsm_{comp}^{1..n}[1..n]$	matrix of integers	/* ts matrix */

```

3.5.1 RW_OPERATION(TYPE, value, r)
3.5.2   abort ← false
3.5.3   op_cnt[r] ← op_cnt[r] + 1
3.5.4   op.id ← (i, op_cnt[r], TYPE, r)
3.5.5   send ⟨SUBMIT, op.id⟩ to server
3.5.6   wait for message ⟨SUBMIT_R, x_op, lso⟩
3.5.7   if not verify(lso.sig) ∧ verify(x_op.sig) then halt
3.5.8   if not ∀k ≠ i : tsm_comp^r[k] ≤ lso.tsm^r[k] ∧ ts_suc[r] = lso.tsm^r[i]
       then halt
3.5.9   if not x_op.id.client_id = r then halt
3.5.10  if not x_op ≤ lso ∧ lso.tsm^r[r] = x_op.tsm^r[r] then halt
3.5.11  forall k = 1..n, k ≠ r do
3.5.12    if not tsm_comp^k, lso.tsm^k are comparable then halt
3.5.13    op.tsm^k ← max{tsm_comp^k, lso.tsm^k}
3.5.14  op.tsm^r ← lso.tsm^r
3.5.15  op.tsm^r[i] ← op_cnt[r]
3.5.16  if TYPE = WRITE then op.value ← value
3.5.17  op.sig ← sign(op.id || op.value || op.tsm)
3.5.18  send ⟨COMMIT, op⟩ to server
3.5.19  wait for message ⟨COMMIT_R, ret_type⟩
3.5.20  tsm_comp ← op.tsm
3.5.21  if ret_type = ABORT then
3.5.22    op.value ← value_suc
3.5.23    abort ← true
3.5.24  else
3.5.25    ts_suc[r] ← op_cnt[r]
3.5.26    value_suc ← op.value
3.5.27    if TYPE = READ then retval ← x_op.value

```

3.6.2 Correctness Arguments

First, we show that all completed operations of client C_i are totally ordered by \leq . This is a reasonable requirement as C_i cannot know if an aborted operation was actually aborted by the malicious server. To achieve this, as the timestamp matrix

of a new operation op of C_i depends on operation lso received in the `SUBMIT_R` message, the check in line 3.5.8 is needed: It guarantees together with lines 3.5.14–3.5.15 that the r th timestamp vector of lso is greater than the one of C_i 's last completed operation stored in tsm_{comp} . For the remaining timestamp vectors it holds by line 3.5.12–3.5.13, as in each case the maximal timestamp vector among lso and tsm_{comp} is picked, that they are greater than the respective one of C_i 's last completed operation. Hence, operation op is greater than the last completed operation of C_i .

Second, we show that when C_i reads value $op_w.value$ from register j during op then op is greater than the corresponding operation op_w with respect to relation \leq . Analogously, by the check in line 3.5.12 and lines 3.5.13–3.5.15, it also holds that op is greater than operation lso . As the check in line 3.5.10 ensures that op_w is smaller or equal than lso , by transitivity, op is greater than op_w . As in the `LINEAR` protocol, when the server returns stale data during a `READ(j)` operation op_i of client C_i , then the `READ` operation and the most recent successful operation of client C_j op_j are forked. In this case the j th timestamp vectors in the matrices of the two operations op_i and op_j are incomparable, especially entry j in the j th timestamp vector of op_i is smaller compared to the corresponding entry in op_j , while the i th entry was updated. The protocol prevents, that any later operation accessing register j of the shared memory is greater than both op_i and op_j . When client C_i accesses register j in a later operation, an operation lso greater than op_j would not pass the check in line 3.5.8. This is because C_i is the only one updating entry i in timestamp vector j , and thus the corresponding entry in lso is smaller than what C_i expects. A similar argument holds for client C_j : If the server presents an operation lso greater than op_i to it, the j th entry of the j th timestamp vector is smaller than what C_j expects and the check in line 3.5.8 is not passed here, too. Operations of other clients accessing register j or operations accessing a different register, as long as all checks are passed are either comparable with op_i or op_j .

These two proof sketches give an intuition how the `CONCUR` protocol ensures that all operations, op causally depends on, are ordered by \leq before op . A detailed correctness proof is given in the next section closing the presentation of the `CONCUR` protocol.

3.6.3 CONCUR Protocol Proof

This section formally proves that the `CONCUR` protocol is an abortable, Byzantine emulation of a shared memory with fork-linearizability. We first define the term *operation*, as used in the `CONCUR` protocol in Algorithm 3.4 and 3.5.

Definition 3.14 (*CONCUR Operation*). An *operation* is a 4-tuple $\langle id, value, tsv, sig \rangle$, where id is an operation id according to Definition 3.4, $value$ is a value from

set \mathcal{V} , tsm is a matrix consisting of n timestamp vectors tsm^1, \dots, tsm^n where each timestamp vector is a vector of size n of integers, and sig is a signature.

As in Section 3.5.3 we define a partial order \leq on operations. Note, that we regard only such operations op after the corresponding timestamp matrix entry $op.tsm$ has been assigned in line 3.5.15 of Algorithm 3.5.

Definition 3.15 (Order Relation). For two operations op and op' holds $op \leq op'$ if and only if

$$\forall i : op.tsm^i \leq op'.tsm^i.$$

Relation \leq on timestamp vectors is given in Definition 3.5. It holds $op = op'$ if and only if op and op' are the same operations.

It is easy to see that \leq relation on operations is transitive. As relation \leq is a partial order on operations, we define a notion of when two operations cannot be ordered by \leq .

Definition 3.16 (Comparable). For two operations op and op' holds op and op' are *comparable* if and only if

$$op \leq op' \vee op' \leq op.$$

Otherwise, they are *incomparable*.

In contrast to the definitions for the LINEAR protocol in section 3.5.3, in the CONCUR protocol clients with incomparable operations are not necessarily forked. Thus, the notion of *forking* is given in the next definition.

Definition 3.17 (Forked). For two operations op and op' holds op and op' are *forked* if and only if

$$\exists i : op.tsm^i \text{ and } op'.tsm^i \text{ are incomparable.}$$

The next Lemma shows that the \leq relation on CONCUR operations does not violate the real-time order of operations.

Lemma 3.18. If $op \leq op'$ then op' does not precede op .

Proof. Let op and op' be two operations of client C_i and C_j and let us assume by contradiction that op' precedes op and $op \leq op'$. During op , client C_i updates the i th entry in the k th² timestamp vector of the timestamp matrix (line 3.5.15). As op' precedes op and as the server cannot forge signatures (line 3.5.17), at the point in time when C_j received the SUBMIT_R message during op' (line 3.5.6), there exists no operation op'' such that $op''.tsm^k[i] \geq op.tsm^k[i]$. Thus, we have that $op.tsm^k[i] > op'.tsm^k[i]$. As op and op' are comparable, this implies that $op.tsm^k > op'.tsm^k$. However, this contradicts the assumption that $op \leq op'$. \square

²W.l.o.g. operation op is an operation that accesses register k of the shared memory.

Analogously to the proof in section 3.5.3, the following two Lemmas show that operations which causally influence each other are ordered by \leq such that the causal order is respected. The operations of one client causally influence each other (Lemma 3.19) as well as a WRITE operation and an operation which reads the written value (Lemma 3.20).

Lemma 3.19. All operations of the same client are totally ordered by \leq relation on operations.

Proof. We show that operation op of client C_i is greater than its previous completed operation op_{comp} . Note, that by line 3.5.20 $op_{comp}.tsm = tsm_{comp}$. By line 3.5.13, as check in line 3.5.12 is passed, we have that $op.tsm^k \geq op_{comp}.tsm^k$ for all $k \neq r$. To pass the check in line 3.5.8, $lso.tsm^r$ is greater or equal than tsm_{comp}^r in all entries but the i th entry. However, in lines 3.5.14 and 3.5.15 the i th entry of the r th vector of the timestamp matrix is updated by a larger entry and we get that $op.tsm^r > op_{comp}.tsm^r$. Thus, we have that $op > op_{comp}$. By induction on C_i 's operations, it follows that op is greater than any operation of C_i that precedes op . \square

Lemma 3.20. If op_r is a READ(j) operation of client C_i that returns $op_w.value$ from the shared memory, then $op_w < op_r$.

Proof. To pass the check in line 3.5.10, it must be that $op_w \leq lso$ and by lines 3.5.13 and 3.5.15 it holds that $lso < op_r$. Thus, if op_r returns $op_w.value$ it must be that $op_w < op_r$. \square

The next definition constructs a sequential permutation of the history of an execution of the CONCUR protocol. The construction helps to simplify the proof of the main correctness proof of the CONCUR protocol.

Definition 3.21 (Sequential Permutation). Let σ be a history of an execution of the CONCUR protocol. We define a *sequential permutation* π of σ by construction: At first we add all events from σ to π . Then, we apply transformations CRASHCOMPLETE and ABORTCOMPLETE (Definition 3.2) in this order to π . Finally, we totally order π by the following rules:

1. The operations are sorted by relation \leq on operations.
2. Yet unsorted operations are sorted according to the real-time order of their completion events in σ .

A subsequence π_i of π contains all operations op of client C_i , and all operations op' that satisfy $op' \leq op$.

In contrast to the LINEAR protocol, during the CONCUR protocol even non-forked clients may produce *incomparable* operations. The next lemma shows how the CONCUR protocol ensures the sequential specification of a shared memory and that forked operations will never be rejoined.

Lemma 3.22. Let op_r be a $\text{READ}(j)$ operation of client C_i that returns $op_w.value$ and op_r is contained in some π_k as defined in Definition 3.21. Then

1. op_w is in π_k , and
2. there is no $\text{WRITE}(j, v')$ operation op'_w of C_j between op_w and op_r in π_k that writes $v' \neq op_w.value$ to the shared memory.

Proof. The first statement follows directly from Lemma 3.20, which states that $op_w < op_r$, and the construction of π_k in Definition 3.21.

For the second statement, note that op'_w and op_w are both operations of client C_j and they write different values. Thus, let us assume for contradiction that such operation op'_w exists and we have that op_w precedes op'_w and op'_w precedes op_r . We first show that (A) op_r and op'_w are forked. Then we show (B) that op'_w is not in π_k .

Proof of A:

We first rule out the trivial case when $i = j$: If $i = j$ then client C_i reads from its own register i . As op_w precedes op'_w we have that at client C_i , $ts_{suc}[i] = op'_w.tsm^i[i] > op_w.tsm^i[i]$. During op_r the check in line 3.5.8 is not passed as $ts_{suc}[i] \neq op_w.tsm^i[i]$ or the check in line 3.5.10 is not passed as $op'_w.tsm^i[i] \neq op_w.tsm^i[i]$. Hence, op_r blocks which contradicts the precondition that op_r is in π_k . Therefore, $i \neq j$.

Let l be operation lso as seen by op_r . To pass the check in line 3.5.10, we have $op_w.tsm^j[j] = l.tsm^j[j]$. As $i \neq j$, client C_i updates only its own entry (line 3.5.15), the j th entry of $l.tsm^j$ is not changed during op_r and thus $op_r.tsm^j[j] = l.tsm^j[j]$ (line 3.5.14). Moreover, since op_w and op'_w are both operations of the same client and op_w precedes op'_w we have $op'_w.tsm^j[j] > op_w.tsm^j[j]$ (lines 3.5.3 and 3.5.15) and thus

$$op_r.tsm^j[j] = op_w.tsm^j[j] < op'_w.tsm^j[j].$$

Further, during op_r the i th entry of the j th timestamp vector is updated to $op_r.tsm^j[i]$. As op'_w precedes op_r and as the server cannot forge signatures, at the point in time when C_j received the SUBMIT_R message during op'_w , there exists no operation op'' such that $op''.tsm^j[i] \geq op_r.tsm^j[i]$. Thus,

$$op'_w.tsm^j[i] < op_r.tsm^j[i]$$

implying that $op_r.tsm^j$ and $op'_w.tsm^j$ are incomparable. Hence, operations op_r and op'_w are forked that proves claim (A) correct.

Proof of B:

To show that op'_w is not included in π_k , we assume by contradiction that op'_w is element of π_k . By construction of π_k there exist minimal operations op , op' of client C_k such that $op_r \leq op$ and $op'_w \leq op'$. As any two operations of C_k are ordered we assume w.l.o.g. that $op' \geq op$ and thus $op' \geq op_r$ and $op' \geq op'_w$. Note that by definition of \leq relation, it must hold for the timestamp vectors that $op'.tsm^j \geq op_r.tsm^j$ and $op'.tsm^j \geq op'_w.tsm^j$. By line 3.5.15, the i th entry of each timestamp vector in a timestamp matrix is only incremented by client C_i . Thus, to satisfy $op'.tsm^j[i] \geq op_r.tsm^j[i]$, there must be a sequence of operations accessing register j , starting with op_r and ending with op' such that the j th timestamp vectors are monotonically increasing.

As the j th entry of each timestamp vector in a timestamp matrix is only incremented by client C_j and $op_r.tsm^j[j] < op'_w.tsm^j[j]$, no operation of client C_j is in this sequence. Otherwise, for client C_j the check ($ts_{suc}[j] = lso.tsm^j[j]$) in line 3.5.8 would not be passed, as the j th entry of the j th timestamp vector is smaller than the corresponding entry of $ts_{suc}[j]$ after op'_w was completed. Thus, as no operation of C_j is in this sequence, all operations in this sequence have $op_r.tsm^j[j]$ as their j th entry in the j th timestamp vector. Therefore, $op'.tsm^j[j] = op_r.tsm^j[j] < op'_w.tsm^j[j]$ and as we have shown that $op'.tsm^j[i] \geq op_r.tsm^j[i] > op'_w.tsm^j[i]$, we conclude that timestamp vectors $op'.tsm^j$ and $op'_w.tsm^j$ are incomparable. This contradicts the fact that $op' \geq op'_w$ and thus, op'_w is not contained in π_k . Hence, also the second statement is satisfied and we are done. \square

The next Lemma proves the main result that the CONCUR protocol satisfies fork-linearizability according to Definition 3.3 on page 38.

Lemma 3.23. The history of any execution of the CONCUR protocol described in Algorithm 3.1, 3.5 and 3.4 is fork-linearizable with respect to the functionality of a shared memory.

Proof. We show that the sequential permutation π of σ and all π_i defined by Definition 3.21 satisfy the properties of fork-linearizability as given in Definition 3.3.

We first show that π maintains real-time order of σ , i.e., if op precedes op' in σ , then op precedes op' in π . By Lemma 3.18, operations sorted by \leq respect real-time order of σ . By the definition of π , all other operations are also ordered in real-time order.

Requirement 2.(a) of fork-linearizability is satisfied by Lemma 3.19, which shows that all operations of one client are totally ordered by \leq , by Lemma 3.20, and by the transitivity of \leq on operations. Requirement 2.(b) follows from Lemma 3.22. Requirement 2.(c) follows directly from the construction of π_i . \square

Lemma 3.10 has shown that the CONCUR protocol satisfies the safety properties of fork-linearizability. We now continue by proving that the CONCUR protocol also satisfies the specified liveness properties. The next lemma proves that as long as server S behaves correctly, all operations eventually complete (by either returning ABORT or OK).

Lemma 3.24. If the server is correct, then no operation in Algorithm 3.5 blocks.

Proof. We have to show that no operation blocks in lines 3.5.7 – 3.5.10.

- $\text{verify}(lso.sig) \wedge \text{verify}(x_op.sig)$ is TRUE: As clients are non-malicious, all signatures are correct. Thus the protocol does not block in line 3.5.7.
- Assume by contradiction that $\exists k : tsm_{comp}^r[k] > lso.tsm^r[k]$: Let op_{comp} be the last completed operation that updated tsm_{comp} in line 3.5.20. As the server is correct it returns only successfully completed operations, so, particularly operations op_{comp} and lso are successful. By line 3.4.12 and 3.4.13 whenever some operation accessing register r of the shared memory takes effect, $lso[r]$ is also updated. Thus, as $lso[r]$ is monotonically increasing, $op_{comp}.tsm^r \not\leq lso.tsm^r$. Therefore, $tsm_{comp}^r \not\leq lso.tsm^r$ which is a contradiction.

Assume by contradiction that $ts_{suc}[r] \neq lso.tsm^r[i]$: Let op_{suc} be the successful operation accessing register r that updated $ts_{suc}[r]$. By the reasoning above, whenever register r at the server is updated, $lso[r]$ is also updated. This implies that $lso.tsm^r[i] \not\leq op_{suc}.tsm^r[i]$. Therefore, it must be that $lso.tsm^r[i] > op_{suc}.tsm^r[i]$. Entry $lso.tsm^r[i]$ is only updated when an operation of C_i later than op_{suc} successfully completes. However, in this case $ts_{suc}[r]$ is also updated to the same value and we have $lso.tsm^r[i] = ts_{suc}[r]$, a contradiction.

Thus, the protocol does not block in line 3.5.8

- $x_op.id.client = r$ is TRUE: As the server is correct it returns x_op from the correct register. Thus the protocol does not block in line 3.5.9.
- Assume by contradiction that $x_op \not\leq lso$: As the server is correct it returns only operations that successfully completed. Thus, both x_op and lso are successful. As x_op and lso both access register r , this implies that x_op and lso can be ordered by \leq . By line 3.4.12 and 3.4.13 whenever some register r at the server is updated, $lso[r]$ is also updated. Thus, as $lso[r]$ is monotonically increasing, $x_op \not\leq lso$. Therefore, $x_op \leq lso$ which is a contradiction.

Assume by contradiction that $lso.tsm^r[r] \neq x_op.tsm^r[r]$: Whenever some register at the server is updated, $lso[r]$ is also updated. This implies that

$lso.tsm^r[r] \not\leq x_op.tsm^r[r]$. Thus, it must be that $lso.tsm^r[r] > x_op.tsm^r[r]$. Entry $lso.tsm^r[r]$ is only updated when an operation of C_r accessing register r later than x_op is successful. However, in this case $x_op.tsm^r[r]$ is also updated to the same value and we have $lso.tsm^r[r] = x_op.tsm^r[r]$, a contradiction.

Thus, the protocol does not block in line 3.5.10

Hence, no operation blocks in lines 3.5.7 – 3.5.10 and we are done. \square

The next lemma shows that also the CONCUR protocol satisfies the *nontriviality* property when server S is correct. This property rules out trivial implementations where every operation is aborted.

Lemma 3.25. With a correct server, an operation op of a client accessing register r aborts only if the server receives a SUBMIT message from another client accessing register r after the SUBMIT and before the COMMIT message corresponding to op .

Proof. If op aborts then the server has received the COMMIT message corresponding to op . As the server is correct and no SUBMIT message of another operation accessing r is received after the SUBMIT and before the COMMIT message of op , $op.id \notin Abrt[r]$. By line 3.4.12 and 3.4.13, op is not aborted. \square

Finally, the following theorem proofs the correctness of the CONCUR protocol.

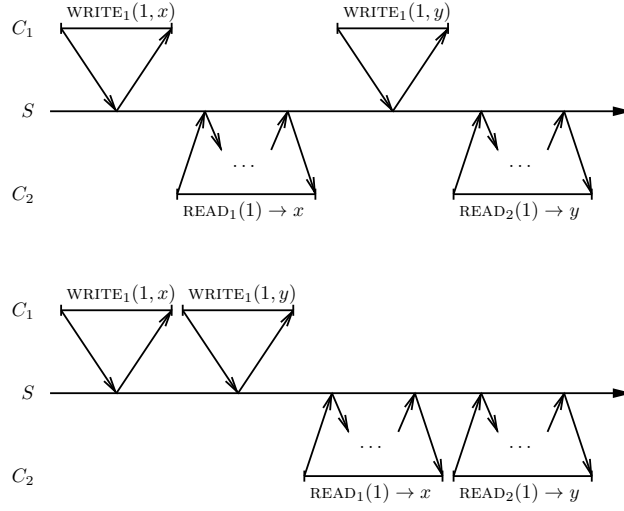
Theorem 3.26. The CONCUR protocol emulates a shared memory on a Byzantine server with fork-linearizability satisfying the properties *nontriviality* and *termination* (cf. Section 3.4).

Proof. By Lemma 3.23, the safety properties of fork-linearizability are satisfied, by Lemma 3.24, the protocol does not block, and by Lemma 3.25, no operation accessing a register in isolation is aborted when the server is correct. \square

3.7 Analysis & Conclusion

In the LINEAR and CONCUR protocol all operations need two communication rounds to complete. We argue why two rounds are necessary for WRITE operations: The reasoning is based on the fact that the information possibly written by some *one*-round WRITE operation is independent from some operations of other clients. Consider the following sequential execution with a correct server and clients C_1 and C_2 : $WRITE_1(1, x)$, $READ_2(1) \rightarrow x^3$, $WRITE_1(1, y)$, $READ_2(1) \rightarrow y$ (Figure 3.1). Note, that by the one-round assumption, the information written by $WRITE_1(1, y)$

³The notation means that $READ_2(1)$ operation of client C_2 returns value x .

Figure 3.1: Two executions, indistinguishable for client C_2

does not depend on the preceding operation $\text{READ}_2(1) \rightarrow x$. Thus, a Byzantine server may “swap” the order of these two operations unnoticeably. Hence, we can construct an execution with a Byzantine server, which is indistinguishable for client C_2 : $\text{WRITE}_1(1, x)$, $\text{WRITE}_1(1, y)$, $\text{READ}_2(1) \rightarrow x$, $\text{READ}_2(1) \rightarrow y$ (see Figure 3.1 for an illustration of the two executions). As C_2 ’s second READ operation returns y , the run violates the sequential specification and thereby also fork-linearizability. Thus, two rounds are needed for WRITE operations and the WRITE operations implemented by the LINEAR and CONCUR protocol are optimal in this sense. The thesis at hand leaves for future work the conjecture that READ operations can be optimized in the LINEAR and CONCUR protocol to complete after a single round. This would also imply that READ operations can be made *wait-free*.

The messages exchanged during the LINEAR protocol have size $\mathcal{O}(2(n + \iota + |v| + \varsigma))$, where ι is the length of an operation id, $|v|$ denotes the maximal length of a value from set \mathcal{V} and ς is the length of a signature. The message complexity of the CONCUR protocol is in $\mathcal{O}(2(n^2 + \iota + |v| + \varsigma))$.

This chapter has introduced two lock-free emulations of fork-linearizable shared memory on a Byzantine server, LINEAR and CONCUR. The LINEAR protocol is based on timestamp vectors and it has a communication complexity of $\mathcal{O}(n)$. It is the first lock-free protocol that emulates fork-linearizable shared memory at all. The impossibility result of Cachin *et al.* [CSS07] is circumvented by aborting concurrent operations. The CONCUR protocol improves on the LINEAR protocol

in the way how concurrent operations are handled. In the CONCUR protocol only concurrent operations accessing the same register of the shared memory need to be aborted. To achieve this, the CONCUR protocol relies on timestamp matrices and has a communication complexity of $\mathcal{O}(n^2)$.

Both protocols demonstrate that in the context of fork-linearizable implementations *obstruction-free* instead of blocking operations can be achieved. This is a major improvement over all existing, lock-based solutions. The main issue when using locks is that client operations are strongly dependent on each other: If a client that currently holds the lock crashes no operations of any other client is able to make progress. Abortable, and thereby obstruction-free operations, as provided by the LINEAR and the CONCUR protocol fundamentally prevent such a strong dependence. Moreover, it has been shown that abortable operations can be easily made wait-free in practical settings [AT08] which ensure maximal independence between the clients.

4 Fork-Consistent Emulations from Registers

The two protocols introduced in this chapter as contributions (C3) and (C4) of this thesis improve on the LINEAR and CONCUR protocol from Chapter 3 (contributions (C1) and (C2)) with respect to the required assumptions on the computational capabilities of the server. The main difference is that in the LINEAR and CONCUR protocol the server is required to execute non-trivial computation steps. From a theoretical viewpoint, such a server constitutes a *universal* object, representing the strongest object in Herlihy’s wait-free hierarchy [Her91]. A universal object is much more powerful than shared storage objects known as *registers*, which lie in the weakest class of all shared objects in this hierarchy. In practical terms, it is important to reduce the complexity and cost of a remote service implementation as computation resources are typically more expensive than storage resources.

This chapter addresses the fundamental structure of fork-consistent implementations and raises the question: Can one provide a fork-consistent emulation in which the server does not execute computation steps, but provides only the functionality of shared registers? Surprisingly, the answer is yes. Specifically, this chapter presents two distributed protocols that are built on top of a server that implements only registers: The AFL protocol (C3) a fork-linearizable emulation of a universal type, in which operations are allowed to abort under concurrency, and the WFL protocol (C4) a weakly fork-linearizable emulation of a shared memory that ensures wait-freedom of the implemented operations as long as the server behaves correctly.

The introduced protocols AFL and WFL constitute the first fork-consistent implementations that are based on a server that implements only registers. Implementing a universal object, as done by the AFL protocol, only from registers has shown to be impossible in a wait-free manner even without fork-consistent semantics [Her91]. The best known construction of a universal type based on registers provides *abortable* operations [AFH⁺07]. As the AFL protocol does not require additional assumptions compared to this approach, the AFL protocol demonstrates that fork-consistent semantics can be added in this context without any trade-offs. The WFL protocol implements a register-based shared memory with fork-consistent semantics where operations are wait-free when the server is correct. All existing solutions require for this task a server that provides *universal* functional-

ity. These approaches appear counterintuitive as the strongest possible server type is required to implement a simple read/write interface of a shared memory. The WFL protocol demonstrates that a shared memory with fork-consistent semantics can be implemented only from memory objects. For instance, the WFL protocol allows to eliminate the server code from the Venus system [SCC⁺10] thereby fundamentally reducing the assumptions made on the computational capabilities of the server.

Section 4.1 introduces the motivation for the taken approach and embeds the contributions in the context of this thesis. Section 4.2 discusses related work and Section 4.3 gives a refinement of the underlying system model. The two main contributions (C3) and (C4) are given as protocols AFL and WFL in Sections 4.4 and 4.5. The chapter concludes with a complexity analysis in Section 4.6.

4.1 Introduction

The increasing trend of executing services online “in the cloud” [MG11] offers many economic advantages, but also raises the challenge of guaranteeing security and strong consistency to its users. As the service is provided by a remote entity that wants to retain its customers, the service usually acts as specified. But online services may fail for various reasons, ranging from simply closing down (corresponding to a crash fault) to deliberate and sometimes malicious behavior (corresponding to a Byzantine fault).

As already discussed in Chapters 1 and 3, cryptographic techniques can prevent a malicious server from forging responses or snooping on customer data. But other violations are still possible, for instance, when multiple isolated clients interact only through a remote server, the latter may send diverging and inconsistent replies to the clients. In this context, “forking” consistency conditions [MS02, CSS07] offer a gracefully degrading solution because they make it much easier for the clients to detect such violations. More precisely, they ensure that if a Byzantine server only *once* sent a wrong response to some client, then this client becomes *forever isolated* or *forked* from those other clients to which the provider responded differently. With this notion, clients may easily detect service misbehavior from a single inconsistent operation, e.g., by out-of-band communication.

Fork-linearizability [MS02, CSS07] ensures that clients always observe linearizable [HW90] service behavior and that two clients, once forked, will never again see each other’s updates to the system (i.e., they share the same history prefix up to the forking point). However, it has been found that fork-linearizable Byzantine emulations of a shared memory *cannot* always provide *wait-free* operations [CSS07], i.e., some clients may be blocked because of other clients that execute operations concurrently. An escape is offered by the weaker liveness property of abortable

emulations as provided by the LINEAR and CONCUR protocol in Chapter 3, which allow client operations to *abort* under contention [MDSS09]. As another alternative, the notion of *weak fork-linearizability* relaxes fork-linearizability in order to allow wait-free client operations in Byzantine emulations [CKS11]. *Weak fork-linearizability* [CKS11] allows two clients, after being forked, to observe a single operation of the other one (at-most-one-join), and that the real-time order induced by linearizability may be violated by the last operation of each client (weak real-time order).

This chapter explores the fundamental assumptions required for building a Byzantine service emulation. Up to now, all fork-consistent protocols have required the server to execute non-trivial computation steps, i.e., the server constitutes an object of *universal* type [Her91], capable of *read-modify-write* operations [KRS88]. Contributions (C3) and (C4) demonstrate the surprising result that this requirement can be dropped, and implement fork-consistent emulations only from memory objects, so-called *registers*, representing one of the weakest forms of computational objects. A long tradition of research has already addressed how to realize powerful abstractions from weaker base objects (e.g., [Her91, AKMS11]).

Specifically, this chapter introduces the *first fork-linearizable Byzantine emulation* of a universal object only *from registers*. The AFL protocol (contribution (C3)) necessarily offers abortable operations because a wait-free emulation of a universal object is not possible in an asynchronous system using only registers [Her91]. Moreover, the WFL protocol (contribution (C4)) is a distributed protocol implementing a *weakly fork-linearizable Byzantine emulation* of a shared memory only from registers. It allows wait-free client operations when the server implementing the underlying registers is correct.

The two protocols AFL and WFL may directly replace the existing respective emulations of shared memory on Byzantine servers [MDSS09, CKS11, SCC⁺10], where the server has *read-modify-write* capabilities. For instance, the WFL protocol, which yields a weakly fork-linearizable Byzantine emulation, allows to eliminate the server code from Venus [SCC⁺10]. Currently, Venus runs server code implemented by a *cloud computing* service, but protocol WFL may realize it from a *cloud storage* service. For practical systems this can make a big difference in cost because full-fledged servers or virtual machines (e.g., Amazon EC2) are typically more expensive than simple disks or cloud-based key-value stores (e.g., Amazon S3).

Contributions of Chapter 4 This chapter proposes, for the first time, Byzantine emulations with fork-consistent semantics only from *registers*, instead of more powerful computation objects. As the registers are implemented by a Byzantine server, any number of registers may be affected by malicious behavior of the server. The introduced protocols are linearizable provided that the server implementing

the base registers are correct. The protocols comprise:

- (C3) The AFL protocol, a register-based abortable Byzantine emulation of a fork-linearizable universal type.
- (C4) The WFL protocol, a register-based wait-free Byzantine emulation of weak fork-linearizable shared memory.

4.2 Related Work

The notion of fork-linearizability was introduced by Mazières and Shasha [MS02]. They implemented a fork-linearizable multi-user storage system called SUNDR. An improved fork-linearizable storage protocol is described by Cachin *et al.* [CSS07]; it reduces the communication complexity compared to SUNDR from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$. More recently, fork-linearizable Byzantine emulations have been extended to *universal services* [Cac11]. All fork-linearizable emulations are blocking and sometimes require one client to wait for another client to complete [CSS07].

In order to circumvent blocking the clients, Majuntke *et al.* [MDSS09] propose the first *abortable* fork-linearizable storage implementations (contributions (C1) and (C2) in Chapter 3). These contributions take up the notion of an abortable object introduced by Aguilera *et al.* [AFH⁺07]. They demonstrated, for the first time, how an abortable (and, hence, obstruction-free [HLM03]) universal object can be constructed from abortable registers, which are shared objects weaker than registers. In more recent work, it has been shown that abortable objects can be boosted to wait-free objects in a partially synchronous system [AT08]. This makes Byzantine emulations of abortable objects, as proposed by contributions (C1), (C2), and (C3) of the thesis at hand, very attractive in practical systems.

Actually implemented systems offering data storage integrity through fork-consistent semantics include SUNDR (LKMS) [LKMS04], which realizes the protocol of Mazières and Shasha [MS02]. Furthermore, Cachin *et al.* [CG09] add fork-linearizable semantics to the Subversion revision control system, such that integrity and consistency of the server can be verified. The “blind stone tablet” of Williams *et al.* [WSS09] provides fork-linearizable semantics for an untrusted database server; it may abort conflicting operations. Using a relaxation of fork-linearizability, called *fork-* consistency*, Feldman *et al.* [FZFF10] introduce a lock-free implementation for online collaboration that protects consistency and integrity of the service against a malicious provider.

Cachin *et al.* [CKS11] present the storage service FAUST, which emulates a shared memory in a wait-free manner by exploiting the notion of *weak fork-linearizability*. It relaxes fork-linearizability in two fundamental ways: (1) after being forked, two clients may observe each others’ operations once more and (2)

the real-time order of the last operation of each client is not preserved. FAUST incorporates client-to-client communication in a higher layer, which ensures that all operations become eventually consistent over time (or the server is detected to misbehave). The Venus system [SCC⁺10] implements the mechanisms behind FAUST and describes a practical solution for ensuring integrity and consistency to the users of cloud storage.

Li and Mazières [LM07] study storage systems, built from $3t + 1$ server replicas, where more than t replicas are Byzantine faulty. Their storage protocol ensures *fork-* consistency*. Similar to weak fork-linearizability, fork-* consistency allows that two forked clients observe again at most one common operation. Standard methods implementing fault-tolerant shared registers from fault-prone base registers show how to *tolerate* up to a fraction of Byzantine base registers [MR98]. This extension, which is orthogonal to the approaches proposed in this chapter and discussed by Chapter 5 of this thesis, would further refine the notion of graceful service degradation with faulty base objects.

4.3 System Model

We consider a distributed system as defined in Section 2.1.1 where the set of clients \mathcal{C} contains $n > 1$ clients C_1, \dots, C_n , and the set of servers contains a single server S .

An execution of a distributed protocol \mathcal{P} induces a history which is a sequence of invocation and response events. For the proposed *abortable* construction (Sec. 4.4), we introduce the special response ABORT. A complete operation o is called *unsuccessful* (“ o is aborted”), if it returns ABORT, else it is called *successful* (“ o successfully completes”). The formal definition of an *abortable* object comprises a non-triviality property which allows aborts only under concurrency [AFH⁺07] (cf. also Section 3.4).

Clients may fail by *crashing*, i.e., they stop taking steps and hence, the last operation of each client might be *incomplete*. Server S may deviate arbitrarily from its specification exhibiting *non-responsive-arbitrary faults* [JCT98] (called *Byzantine*). Clients have access to a digital signature scheme used by each client to *sign* its data such that any other client can determine the authenticity of a datum by *verifying* the corresponding signature. We assume that signatures cannot be forged.

We omit the algorithm of server S and assume that server S implements a collection of *atomic registers* that can be directly accessed by the clients (see Section 2.1.4 for a formal definition). An atomic register provides two operations, *read* and *write*¹. Operation *write*(v) stores value v into the register. A call of

¹We type operation calls to base registers provided by server S in *italic* font and calls to the

$read()$ returns the latest written value from the register or the special value \perp if no value has been written. As the register is atomic, its history satisfies linearizability [Her91], i.e., operations seem to appear as sequential, atomic events². Further, the atomic registers used allow single-writer-multiple-reader access (SWMR), i.e., to each register we assign a dedicated client that may call *write* and *read*, while all other clients may only call *read* to that register.

A sequence of operations π satisfies *weak real-time* order of history σ if π , excluding the last operation of each client in π , satisfies real-time order of σ . *Causality* between two operations depends on the type of the implemented object³. For two operations of a shared memory o and o' in σ , o *causally precedes* o' ($o \rightarrow_\sigma o'$), if o and o' are called by the same client and o happens before o' , or if o' is a READ operation that returns the value written by WRITE operation o :

Definition 4.1. For two operations o, o' in history σ of an execution of a distributed protocol implementing a shared memory, we say that o *causally precedes* o' in σ (o' *causally depends on* o), denoted $o \rightarrow_\sigma o'$ whenever one of the following conditions hold:

1. Operations o and o' are both invoked by the same client and o finishes before o' is invoked.
2. operation o' is a READ operation, o is a WRITE operation, and o' reads the value written by o .
3. There exists an operation o'' such that $o \rightarrow_\sigma o''$ and $o'' \rightarrow_\sigma o'$.

Before defining *(weak) fork-linearizability*, we first formalize the notion of a *possible view* [CKS11] and the *weak real-time order* [CKS11].

Definition 4.2. A sequence of events π is called a *possible view* of a history σ at a client C_i with respect to a functionality F if σ can be extended (by appending zero or more responses) to a history σ' such that:

1. π is a sequential permutation of some subsequence of $complete(\sigma')$,
2. $\pi|_{C_i} = complete(\sigma')|_{C_i}$, and
3. π satisfies the sequential specification of F .

Where for a sequence of events σ , $complete(\sigma)$ is the maximal subsequence of σ consisting only of complete operations.

implemented shared objects in CAPITALS.

²Hence, the “latest written value” is well-defined.

³As causality is needed to define *weak fork-linearizability*, here, we give causality for a *shared memory*, which is the type we implement with weak fork-linearizability.

Definition 4.3. Let π be a sequence of events and let $\text{lastops}(\pi)$ be a function of π returning the set containing the last operation from every client in π (if it exists), that is,

$$\text{lastops}(\pi) := \bigcup_{i=1, \dots, n} \{o \in \pi|_{C_i} \mid \nexists o' \in \pi|_{C_i} \text{ s.t. } o \text{ precedes } o' \text{ in } \pi\}$$

We say that π preserves the *weak real-time order* of a sequence of operations σ whenever π excluding all events belonging to operations in $\text{lastops}(\pi)$ preserves the real-time order of π .

The next definition formalizes the notion of *fork-linearizability* [CSS07] and *weak fork-linearizability* [CKS11]. The definition of *fork-linearizability* is equivalent to the corresponding definitions given in Chapters 2 and 3.

Definition 4.4. Let σ be a history of an execution of distributed protocol \mathcal{P} implementing functionality F and for each client C_i there exists a sequence of events π_i such that π_i is a possible view of σ at C_i with respect to F .

History σ is *fork-linearizable* with respect to functionality F if for each client C_i :

1. π_i preserves the real-time order of σ , and
2. for every client C_j and for every $o \in \pi_i \cap \pi_j$, it holds $\pi_i|_o = \pi_j|_o$.

History σ is *weak fork-linearizable* with respect to functionality F if for each client C_i :

1. π_i preserves the weak real-time order of σ , and
2. for every operation $o \in \pi_i$ and every operation $o' \in \sigma$ such that $o' \rightarrow_\sigma o$, it holds that $o' \in \pi_i$ and that $o' <_{\pi_i} o$, and
3. (At-most-one-join) for every client C_j and every two operations $o, o' \in \pi_i \cap \pi_j$ by the same client such that $o <_\sigma o'$, it holds $\pi_i|_o = \pi_j|_o$.

The notion of a *Byzantine emulation* [CSS07] as given in Definition 2.4 in Chapter 2 comprises the safety and liveness properties of the proposed distributed protocols. Note that the liveness condition of abortable operations is weaker than *wait-freedom* but still not weaker than *obstruction-freedom* [AFH⁺07]. Such a Byzantine emulation is *wait-free* (*abortable* resp.), iff every fair and well-formed execution of the protocol with a correct server is wait-free [Her91] (*abortable* [AFH⁺07] resp.).

4.4 (C3): The AFL Protocol

This section presents as contribution (C3) of this thesis the AFL protocol which is an abortable fork-linearizable Byzantine emulation of a universal type implemented on top of a server providing only atomic registers. The shared functionality ensures fork-linearizability in the presence of any number of faulty base registers. High-level operations are *abortable* [AFH⁺07], i.e., under concurrency, the special response ABORT may be returned. The functionality of a universal type T is encoded in the procedure APPLY_T . For client C_i , state s and instruction ins , $\text{APPLY}_T(s, ins, i)$ returns (s', res) , where s' is the new state of the universal object, res the computation result, and where the sequence of invoking $\text{APPLY}_T(s, ins, i)$ and returning (s', res) is defined by the sequential specification of type T (see Section 2.1.3 for a formal definition).

The AFL protocol uses timestamp vectors called *versions* whose order reflects the real-time order in which operations are applied to the implemented shared functionality. Each operation carries a version, and the linearization of operations is achieved through the use of an INC&READ counter object C with two atomic operations INC&READ and READ. An invocation to INC&READ(C) advances the counter object C and returns a value which is higher than any value returned before, and READ(C) returns the current value of the counter object. An implementation of the INC&READ counter is given in Algorithm 4.2 in Section 4.4.4 together with its formal properties. This algorithm uses wait-free atomic registers (implemented by server S) as base objects which makes it a wait-free variant of the *abortable* INC&READ counter described by Aguilera *et al.* [AFH⁺07].

4.4.1 Protocol Ideas

Universal Type To implement universal type T , the AFL protocol uses n SWMR registers R_1, \dots, R_n provided by server S such that client C_i can read from all registers but may write only to R_i . The registers store states of the implemented universal object. To implement high-level operations, client C_i reads from the register which holds the most current state, applies the relevant state transformation, and writes the new state to R_i . Note, that all information are digitally signed by the clients as base objects are untrusted. Thereby, operations “affect” each other which leads to the following relation on operations: Operation o of C_i *affects* operation o' of C_j , if during o' , C_j is able to verify the signature of C_i on state s that has been written during o and if C_j executes APPLY_T on s during o' ; further, an operation of C_i *affects* each later operation of C_i .

Concurrency detection Operations of the AFL protocol are allowed to abort under concurrency for two reasons: there is no wait-free construction of a universal

type from registers, as shown by Herlihy [Her91], and no fork-linearizable protocol can be wait-free in all executions, as shown in a more recent work of Cachin *et al.* [CSS07]. Cachin’s impossibility is based on two runs, indistinguishable for the reader: In the first run a READ operation does not return value v as it is concurrently written, while in the second run v has been previously written and is hidden by malicious registers (see Figure 1.1 on page 4 in Chapter 1 for an illustration).

To avoid such a situation, the AFL protocol implements a concurrency detection mechanism [AFH⁺07] using INC&READ counter object C . If concurrency is detected, a pending operation is aborted. At the invocation of a high-level operation o , the AFL protocol calls INC&READ(C) and stores the timestamp returned. At the end of o , READ(C) is executed to check whether counter C still returns the same timestamp. If not, another operation o' was invoked during o — thus, o is aborted. Else, if at the end of o C has not been changed, all successful operations either terminated before o or will be invoked after o has terminated. This is because the timestamps, returned from INC&READ, are used to linearize operations: The current state is written together with the timestamp, and the timestamp is used to determine the most recent state. Hence, all other operations invoked so far write a state with a lower timestamp than o . Consequently, such operations are linearized before operation o and only the state written by o can be read by later operations.

Fork-Linearizability In addition to the timestamp from INC&READ counter C , each operation is assigned a vector of timestamps of length n , called *version*. The order relation \leq defined on versions respects real-time order and the ”affected by” relation on operations. The idea is that each operation reads the most recent version from the registers, increments its own entry and writes the new version back to the registers. Thereby, each operation checks, if the version it reads, has been affected by the version of its own last successful operation, i.e., one which was not aborted. If the last successful operation of client C_i is hidden from C_j , then C_i does not accept operations of C_j as they have *not* been affected by the last successful operation of C_i . This ensures that the views of the clients after a forking attack are not rejoined. This principle is based on ideas of Mazières and Shasha [MS02], and Cachin *et al.* [CSS07] and has also been used in a similar way by the LINEAR and CONCUR protocol in Chapter 3.

To apply this idea to the AFL protocol, we have to add a specific handling for aborted operations: If operation o of client C_i is aborted, C_i cannot expect that o will affect later operations. However, it is still possible that some operation of C_j is affected by aborted o . In this case we call o *relevant* for C_j (Definition 4.6 on page 75).

4.4.2 Description of the AFL Protocol

In the following, the steps performed by client C_i when executing high-level instruction ins of protocol AFL are described. The AFL protocol is given as Algorithm 4.1.

The algorithm executed by client C_i is framed by $INC\&READ(C)$ and $READ(C)$ calls to the counter object C implementing the concurrency detection mechanism (lines 4.1.2 and 4.1.14). If the returned timestamps are not equal, the operation is aborted in line 4.1.16. In lines 4.1.3–4.1.5, the client reads from all atomic registers R_1, \dots, R_n and determines by means of the assigned timestamps the index l of the register holding the latest written data $\langle ts_l, V_l, s_l, sig_l \rangle$, where ts_l is a timestamp, V_l is the version, s_l is the state and sig_l is a signature. If some data have been written to R_l , the signature of the content of R_l is verified (line 4.1.6). Then, client C_i checks whether the read version V_l is not smaller than V_{suc} the version of its own last successful operation (line 4.1.7). When the check is passed the new state of the universal object and the computation result is computed by calling $APPLY_T(s_l, ins, i)$ (line 4.1.8). Finally the new version of the operation has to be computed. This is done by taking the per-entry maximum of version V , which is the local version of C_i , and V_l , and by incrementing the i th entry (lines 4.1.9–4.1.11). After signing the current timestamp, the new version V , and new state s in line 4.1.12, client C_i writes ts , V , s and the signature into register R_i (line 4.1.13). If operation o is successful, version V is stored as last successful version V_{suc} and the computation result is returned (lines 4.1.17–4.1.19).

4.4.3 Correctness Arguments

In this section we argue why the AFL protocol in Algorithm 4.1 satisfies fork-linearizability. The goal is to construct for each client C_i a view π_i of history σ that satisfies the properties of fork-linearizability. To construct π_i , we simplify our argumentation by ignoring operations that are not relevant for C_i . Recall, any operation is *relevant* for client C_i that affects C_i 's last successful operation. Hence, operations that are not relevant for client C_i do not change the object's state from C_i 's point of view. Thus, we can order them arbitrarily among the operations in π_i and the resulting sequences still satisfy fork-linearizability.

The idea behind the construction of the π_i in the proof is that operations are ordered according to their assigned versions. The proof shows that this order respects the “affected by” relation, the sequential specification of a universal type, and the real-time order. As during an operation the new version is computed using the client's last version and the read version, proving “affected by” and real-time order is straightforward. The core of the proof is to show that the order of version also respects the sequential specification. We sketch the intuition behind this with

Algorithm 4.1: AFL Protocol, Algorithm of Client i

Variables:
 C INC&READ counter object, initially 0
 R_1, \dots, R_n SWMR atomic register, initially $\langle 0, (0, \dots, 0), \perp, \perp \rangle$
 ts, ts', ts_l, cn integer, initially 0
 $V[1..n], V_l[1..n], V_{suc}[1..n]$ array of integers, initially $(0, \dots, 0)$
 s, s_l state, initially \perp
 res operation result, initially \perp
 sig, sig_l signature, initially \perp

```

4.1.1 EXECUTE( $ins$ ) do
4.1.2    $ts \leftarrow \text{INC\&READ}(C)$  /* increment and read from counter */
4.1.3   for  $j = 1, \dots, n$  do
4.1.4      $\langle ts_j, V_j, s_j, sig_j \rangle \leftarrow \text{read}(R_j)$  /* low-level atomic read */
4.1.5     let  $l$  be such that  $ts_l = \max_{1 \leq j \leq n}(ts_j)$  /* register with newest data */
4.1.6     if  $V_l \neq [0 \dots 0] \wedge \neg \text{verify}_l(sig_l, \langle ts_l, V_l, s_l \rangle)$  then halt /* signature ok? */
4.1.7     if  $\exists k : V_{suc}[k] > V_l[k]$  then halt /* fork-lin check passed? */
4.1.8      $\langle s, res \rangle \leftarrow \text{APPLY}_T(s_l, ins, i)$  /* compute new state + result */
4.1.9     for  $j = 1, \dots, n, j \neq i$  do
4.1.10       $V[j] \leftarrow \max(V[j], V_l[j])$  /* determine
4.1.11      new version */
4.1.11       $V[i] \leftarrow V[i] + 1$ 
4.1.12       $sig \leftarrow \text{sign}_i(ts || V || s)$  /* signature on ts, version, state */
4.1.13       $\text{write}(R_i, \langle ts, V, s, sig \rangle)$  /* low-level atomic write */
4.1.14       $ts' \leftarrow \text{READ}(C)$  /* read from counter */
4.1.15      if  $ts \neq ts'$  then
4.1.16        return ABORT /* concurrency detected */
4.1.17      else
4.1.18         $V_{suc} \leftarrow V$  /* reset last successful version */
4.1.19      return  $res$  /* return result */

```

the following argumentation leading to a contradiction:

Assume that some operation o_c is not affected by the most recent state of the universal object, which has been written by relevant operation o_b , but is affected by an older state written by operation o_a . In this case, the clients of o_b and o_c are forked, and neither o_b nor o_c affect each other. We argue, that in such a situation, there is no relevant operation that has been affected by both o_b and o_c , as such an operation would join the two clients violating fork-consistency. We assume for contradiction, that a relevant operation o_{join} of client C_{join} , affected by o_b and o_c exists which is also the first among such operations (see Figure 4.1). Operation o_{join} is affected by o_{join_suc} , the last successful operation of C_{join} previous to o_{join} , and by o_r that wrote the state which is read during o_{join} . Hence, without loss of generality o_{join_suc} is affected by o_b while o_r is affected by o_c . During operation

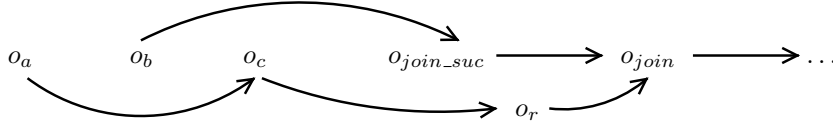


Figure 4.1: Correctness Idea of the AFL Protocol in Algorithm 4.1. Arrows denote the “affected by” relation.

o_{join_suc} , client C_{join} raises its value in the version to $V[join]_{join_suc}$. This implies that o_{join} only accepts versions where the *jointh* entry is at least $V[join]_{join_suc}$ (line 4.1.7). As o_{join_suc} is not on the path of “affected by” relations from o_c to o_r , o_{join} would block while reading the state of o_r which is a contradiction. Thus, o_{join} does not exist.

Finally, it follows directly from the described construction, that sequences π_i satisfy the no-join property. To complete the correctness proof of the Byzantine emulation, we show that when all base objects are correct, no operation blocks and that no operation trivially aborts.

4.4.4 Proof of Correctness of the AFL Protocol

This section formally proves that the AFL protocol given as Algorithm 4.1 implements an abortable, fork-linearizable Byzantine emulation of a universal type.

The implementation uses an INC&READ counter object, given as Algorithm 4.2 for client C_i , $i = 1, \dots, n$. An INC&READ counter object C provides two atomic operations INC&READ(C) and READ(C). An invocation to INC&READ(C) advances the counter object C and returns a value which is higher than any value returned before the invocation of INC&READ(C). An invocation to READ(C) returns the current value of the counter object. The INC&READ counter C has two properties:

- P1** If a client process runs in isolation and it first calls INC&READ(C) and then later READ(C), then the same value is returned by both invocations, and
- P2** the values returned by INC&READ(C) reflect the real-time order of invocations to INC&READ(C).

The counter object C is a wait-free variant of the abortable INC&READ counter described by Aguilera *et al.* [AFH⁺07]. For the implementation of the INC&READ counter, instead of abortable base registers [AFH⁺07], wait-free atomic registers provided by server S are used here, hence the counter does not need to abort.

We further define the “affected by” relation of two (high-level) operations implemented by the AFL protocol (Definition 4.5), the notion of relevant operations (Definitions 4.6 and 4.7), and the \leq order relation on versions (Definition 4.8).

Algorithm 4.2: INC&READ Counter for n Clients, Algorithm of Client C_i

Variables:
 R_1, \dots, R_n , SWMR atomic registers, initially 0
 $cnt_1, \dots, cnt_n, k, c, id$, integers, initially 0

4.2.1 INC&READ() **do**
4.2.2 **for** $k = 1, \dots, n$ **do** $cnt_k \leftarrow read(R_k)$
4.2.3 $c \leftarrow \max_{1 \leq k \leq n} \{cnt_k\} + 1$
4.2.4 $write(R_i, c)$
4.2.5 **return** $n \cdot c + (i - 1)$

4.2.6 READ() **do**
4.2.7 **for** $k = 1, \dots, n$ **do** $cnt_k \leftarrow read(R_k)$
4.2.8 $c \leftarrow \max_{1 \leq k \leq n} \{cnt_k\}$
4.2.9 $id \leftarrow \max_{1 \leq k \leq n} \{k | cnt_k = c\}$
4.2.10 **return** $n \cdot c + (id - 1)$

Definition 4.5. For two operations o, o' in a history σ of an execution of the AFL protocol we say that o *affects* o' in σ (o' *is affected by* o) whenever one of the following conditions holds:

1. Operations o and o' are both invoked by the same client, o is successful and o finishes before o' is invoked.
2. Operation o' reads the state s_l (version V_l) written by o , successfully verifies the signature and executes $APPLY_T$ to s_l (during o' in lines (4.1.5—4.1.10 of Algorithm 4.1), V_l is the version, s_l the state, ts_l the timestamp and sig_l the signature written during o).
3. There exists an operation o'' such that o *affects* o'' and o'' *affects* o' .

The notion of a *relevant* operation is defined recursively by the next definition.

Definition 4.6. An operation o is *relevant* if and only if

1. o is *successful* OR
2. there exists a relevant operation o' that has been affected by o .

Definition 4.7. An operation o is *relevant* for client C_i if and only if some successful operation of C_i has been affected by o .

Definition 4.8 (Order Relation). A *version* V is a vector of integers of length n , initially $(0, \dots, 0)$. For two versions V and V' holds $V \leq V'$ if and only if

$$\forall i : V[i] \leq V'[i].$$

It holds $V = V'$ if and only if V and V' are the same versions.

For two operations o and o' with versions V and V' holds $o \leq o'$ if and only if

$$V \leq V'$$

It holds $o = o'$ if and only if o and o' are the same operations.

It is easy to see that \leq relation on operations (versions) is *transitive*. The next definition introduces the notion of operations taking *effect*. Note, that the last operation of each client, when the client crashes, may be incomplete but may appear as a complete operation to others — i.e., it took effect.

Definition 4.9. An operation of client C_i *takes effect* if and only if the low-level *write* operation in line 4.1.13 successfully returns.

The next Corollary shows that \leq relation on operations respects the real-time order of sequential operations.

Corollary 4.10. If o and o' are two operations and $o \leq o'$ then operation o' does not precede operation o .

Proof. Let o and o' have associated versions V and V' respectively. Assume by contradiction that o' precedes o and that $o \leq o'$. During o , the entry $V[i]$ is incremented. As o' precedes o and as versions are digitally signed (line 4.1.12), it holds that $V'[i] < V[i]$. Hence, $V' \not\geq V$ and therefore $o \not\leq o'$. \square

The following two Corollaries show that operations which affect each other are ordered by \leq such that the “affected by” relation is respected. According to Definition 4.5, the successful operations of one client affect each other (Corollary 4.11) as well as an operation that is applied to the state updated by another operation (Corollary 4.12).

Corollary 4.11. All operations of the same client are totally ordered by \leq relation on operations.

Proof. We show that operation o' of client C_i is greater than its previous completed operation o . Let V and V' be the versions of operation o and o' respectively. Let V_l be the version read by operation o' . The entries $V'[k]$, $k \neq i$ is assigned the maximum of $V_l[k]$ and $V[k]$ (line 4.1.10), and $V'[i]$ is updated with a value larger than $V[i]$, as $V[i]$ is incremented with every invoked operation of C_i (line 4.1.11). Clearly, $V' > V$. By induction on C_i 's operations, it follows that o' is greater than any preceding operation of C_i . \square

Corollary 4.12. If o' is reading state s from some register R_i updated by operation o , then $o > o'$.

Proof. Let V and V' be the versions of operations o and o' respectively. When operation o' is applied to state s , then V is version V_l during operation o' . By lines 4.1.10–4.1.11 and analogously to the proof of Corollary 4.11 it follows directly that $V' > V$. \square

The following Lemma shows the main result of this section: The universal type, implemented by the AFL protocol satisfies fork-linearizability (Definition 4.4 on page 69). The proof shows how for each client the subsequences π_i are constructed. Then, by proving two claims, we show that sequences π satisfy the properties of fork-linearizability. To ease the argumentation, operations which are not relevant at all or not relevant for client C_i are ignored.

Lemma 4.13. The history σ induced by any execution of the AFL protocol (Algorithm 4.1) satisfies fork-linearizability with respect to a shared functionality of a universal object with type T .

Proof. Let σ be the history of an execution of the AFL protocol. We first remove all invocations of incomplete operations that do not take effect (Definition 4.9). We add the corresponding completion event of incomplete operations that take effect directly at the end of σ . Then we remove all operations that are not relevant. Note, that σ now contains only complete and relevant operations.

We construct a sequential permutation π by totally ordering all events in σ . To achieve this, we order the events in σ by the following rules:

1. Sort the operations in σ by \leq relation on operations.
2. Sort any yet unsorted operations by the real-time order of their completion event.

We construct the subsequences π_i (for $i = 1, \dots, n$) as required by the definition of fork-linearizability (Definition 4.4). We include in π_i all operations of client C_i in π . Then, for all $o \in \pi_i$ we include into π_i all operations o' in π such that $o' \leq o$. Finally, we remove all operations that are not *relevant* for C_i .

By Corollary 4.10, as \leq relation on operations respects real-time order, the following claim follows directly:

Claim 13.1 Let o and o' be two operations and o precedes o' in σ . Then, o precedes o' in π .

Claim 13.2 Let o_c be an operation of client C_i in sequence π_m of client C_m , $m \in 1, \dots, n$, that updates state s in register R_j ; state s was written by operation o_a of client C_j , $j \in 1, \dots, n$, into register R_j . Then:

1. Operation o_a is in π_m , and

2. in π_m there is no operation by client C_k , $k \in 1, \dots, n$, that is *relevant* for C_m , that is subsequent to o_a in π_m , and that completes before o_c is invoked.

By Corollary 4.12 holds that $o_c > o_a$. Hence, o_a is included in π_m by construction and the first statement of Claim 13.2 follows directly.

To prove the second statement of Claim 13.2, let us assume for contradiction that such an operation o_b of client C_k exists in π_m which is invoked after o_a completes and which completes before o_c is invoked. Hence, o_a, o_b, o_c are three sequential operations in that order in σ .

We first show that o_b and o_c do not affect each other, i.e., o_b is not affected by o_c and o_c is not affected by o_b :

- “ o_c is not affected by o_b ”: Operation o_c is affected by o_a and by o_{suc} , the last successful operation by client C_i previous to o_c , if it exists. If o_{suc} is affected by o_b then o_a precedes o_{suc} . Hence, the i th entry in the version of o_{suc} is greater than the one of o_a and therefore o_a could not affect o_c (as the check in line 4.1.7 is not passed) — a contradiction. If o_{suc} is not affected by o_b , then o_c is also not affected by o_b (as “affected by” relation is transitive; Definition 4.5).
- “ o_b is not affected by o_c ”: Follows directly as o_b precedes o_c .

Next, we derive a contradiction to the assumption that operation o_b exists. As operations o_b and o_c are both relevant for client C_m and o_c and o_b do not affect each other, there are successful operations o'_b and o'_c of client C_m such that o'_b is affected by o_b and o'_c is affected by o_c . Let o'_b and o'_c be the operations of C_m that are affected by o_b or o_c respectively with the smallest versions (they exist by Corollary 4.11). Note that, as o'_b and o'_c are both successful operations of the same client C_m , they affect each other. Let us assume w.l.o.g. that o'_c is affected by o'_b . This means, there exists some operation o_{join} of client C_{join} , $join \in 1, \dots, n$, which is the operation with the smallest timestamp that is affected by both o_b and o_c . For operation o_{join} either holds

- (A) $o_{join} \leq o'_b$ and $o'_b (= o'_c)$ is affected by o_{join} , or
- (B) o_{join} is affected by o'_b and $o'_c (\neq o'_b)$ is affected by o_{join} .

To have o_{join} to be affected by two operations that do not affect each other, by Definition 4.5, (1) there must be some operation o_{join_suc} which is the last successful operation of client C_{join} previous to o_{join} that is affected either by o_b or o_c . W.l.o.g. we assume that o_{join_suc} is affected by o_b . Note, that $o_{join_suc} \geq o_b$. Further, o_{join_suc} is not affected by o_c , as otherwise o_{join} would not be the first

operation affected by both o_b and o_c ($o_{join_suc} < o_{join}$ by Corollary 4.11). Further, (2) o_{join} reads the state written by some operation o_r that is affected by o_c and $o_r \geq o_c$. Analogously, o_r is not affected by o_b ($o_r < o_{join}$ by Corollary 4.12). Hence, as o_b and o_c do not affect each other, there are disjunct “affected by” paths⁴ from o_b to o_{join_suc} and from o_c to o_r (Figure 4.2).

During operation o_{join_suc} the *jointh* entry of its version is incremented to $V_{join_suc}[join]$ (line 4.1.11) and as o_{join_suc} is successful, it also holds $V_{suc}[join] \geq V_{join_suc}[join]$ (line 4.1.18) from this point on at client C_{join} . Consequently, during o_{join} , as o_{join_suc} precedes o_{join} , client C_{join} does not accept version V_l such that $V_l[join] < V_{join_suc}[join]$ (check in line 4.1.7 would not be passed). Hence, there must be an operation o'_{join} on the path from o_c to o_r that raises the *jointh* entry in the versions to $V_{join_suc}[join]$ or higher. Note, as the *jointh* entry is only raised by an operation of C_{join} (line 4.1.11), o'_{join} has to be an operation of C_{join} as well. Operation o'_{join} cannot precede o_{join_suc} , as o_{join_suc} is the first operation to raise the *jointh* entry to $V_{join_suc}[join]$ (line 4.1.11). If o'_{join} follows o_{join_suc} , then o'_{join} does not accept versions V_l with $V_l[join] < V_{join_suc}[join]$ (check in line 4.1.7), as $V_{suc}[join] \geq V_{join_suc}[join]$ at C_{join} after o_{join_suc} has finished. Hence, o'_{join} does not exist on the path from o_c to o_r and o_{join} would block when reading the version of o_r . Thus, we have a contradiction and o_{join} does not exist.

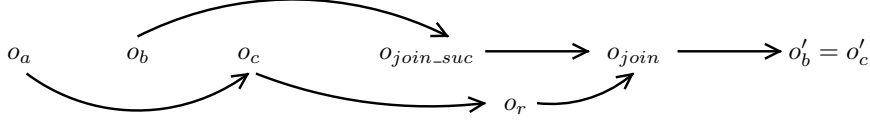
This means that either o'_b does not exist (case A) which implies that o_b is not relevant for C_m or o'_c does not exist (case B) which implies that o_c is not relevant for C_m . Consequently, the assumption that there is an operation o_b between o_a and o_c in π_m is wrong and thus, all operations in π_m are totally ordered by \leq .

We now show that π_m for all $m = 1, \dots, n$ satisfies fork-linearizability as given in Definition 4.4. To show that π_m is a possible view of client C_m , properties 1. and 2. of Definition 4.2 follow directly from the construction of π_m given at the beginning of the Lemma. Claim 13.2 proves property 3. of Definition 4.2. Hence, π_m is a possible view of client C_m . Each sequence π_m satisfies real-time ordering as shown in claim 13.1. The no-join property (condition 3. in Definition 4.4) is also an easy consequence of the construction of π_m . The non-relevant operations that have been removed at the beginning of this proof, can be added to all π_m in real-time order of their completion events. As they are not relevant, they do not effect the sequential specification and thus, they do not violate fork-linearizability. \square

The next two Lemmas show that the AFL protocol implements an abortable Byzantine emulation with fork-linearizability of a universal type (see Definition 2.4 on page 30). Lemma 4.14 shows that no operation blocks, and Lemma 4.15 proves that no operation is trivially aborted.

⁴An “affected by” path from operation o_1 to o_x is a sequence of operations o_1, o_2, \dots, o_x such that for $i = 1, \dots, x-1$, o_i affects o_{i+1} .

Case A



Case B

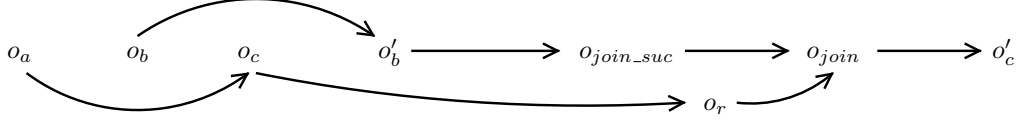


Figure 4.2: Proof of Lemma 4.13. The arrows denote the “affected by” relation between operations.

Lemma 4.14. If registers R_i, \dots, R_n and INC&READ object C are implemented by a correct server S , and σ is the history induced by any execution of the AFL protocol, then no operation in σ halts in line 4.1.6 nor in line 4.1.7 of Algorithm 4.1.

Proof. We show that no operation in Algorithm 4.1 blocks: If the base objects are correct, and as clients are trusted, no signature is forged and thus no operation blocks in line 4.1.6.

It remains to show that no operation of client C_i blocks in line 4.1.7. Assume by contradiction that during operation o of client C_i $\exists k : V_{suc}[k] > V_l[k]$. Let o_{suc} be the successful operation of C_i that wrote some state with version V_{suc} to register R_i . As o_{suc} is not aborted, client C_i has read the same timestamp from the INC&READ object C in line 4.1.2 and 4.1.14 during o_{suc} . This means, as the INC&READ object C is correct, that no other operation executed INC&READ(C) between INC&READ(C) and READ(C) of o_{suc} and that o_{suc} has written the highest timestamp so far (line 4.1.13). Hence, INC&READ(C) of operation o_l , that wrote version V_l , happened either (1) before INC&READ(C) of o_{suc} or (2) after READ(C) of o_{suc} . In case (1) o would not find l as the highest index in line 4.1.5 and thus it would not read o_l , as o_{suc} holds a higher timestamp than o_l (INC&READ object is correct) — a contradiction. For case (2), o_l would read a version $\geq V_{suc}$ and thus $V_l \geq V_{suc}$ (line 4.1.10) — a contradiction. Concluding, during operation o of client C_i no such $k : V_{suc}[k] > V_l[k]$ exists and thus, the protocol does not block in line 4.1.7. \square

Lemma 4.15. If registers R_i, \dots, R_n and INC&READ object C are implemented by a correct server S then, if operation o in an execution of the AFL protocol returns ABORT, then o is concurrent with some other operation.

Proof. The correctness follows directly from the properties of INC&READ object C : Operation o is only aborted if the condition in line 4.1.15 is satisfied. This is the case when object C returns a different value to call in line 4.1.14 than in line 4.1.2. The properties of C imply, that this happens only if some other operations calls INC&READ(C) in the meanwhile. This means, some other operation is concurrent with o (according to the definition in Section 4.3 on page 67) and thus, we are done. \square

Note, that Lemma 4.15 is sufficient to show that the AFL protocol implements an *abortable* object. It is easy to see that in every situation where an operation of Algorithm 4.1 aborts, Aguilera's universal type construction ([AFH⁺07], Algorithm 2) would abort as well.

Finally, the correctness of the AFL protocol has been shown in Lemma 4.13 (Fork-Linearizability), Lemma 4.14 (No Blocking), and Lemma 4.15 (Nontriviality).

4.5 (C4): The WFL Protocol

This section introduces the WFL protocol (contribution (C4) of the thesis) which is a wait-free, weak fork-linearizable Byzantine emulation of a shared memory implemented from a server that provides only atomic registers. The WFL protocol satisfies weak fork-linearizability in the presence of any number of faulty base registers. The implemented shared memory provides n atomic registers, such that each client can write to one dedicated register exclusively and may read from all registers. Operation $\text{WRITE}(i, v)$, called by client C_i , writes value v to C_i 's register. Operation $\text{READ}(i)$ returns the last written value from C_i 's register, and may be called by any client. The WFL protocol makes use of an atomic single-writer snapshot object SO with n components [AGR08, Fic05]. Snapshot object SO provides two atomic operations: $\text{UPDATE}(d, SO, i)$, that changes the state of component i of SO to d , and $\text{SCAN}(SO)$ that returns vector (d_1, \dots, d_n) such that d_i is the state of component i of SO , $i = 1 \dots n$. Formally, d_i is the state written by the last UPDATE to component i prior to SCAN . It has been shown, that such a shared snapshot object can be wait-free implemented only from registers [AGR08, Fic05], and thus we assume that SO is also provided by server S .

4.5.1 Protocol Ideas

Each client locally maintains a timestamp that respects causality and real-time order of its *own* operations. As the basic principle, during each operation this timestamp is written to the shared memory and timestamps left by other operations are read. For each client C_i the WFL protocol uses two registers only C_i

client checks during o_r whether the next WRITE operation after o_w (of the same client as o_w), has seen READ operation o_r or a newer one. Else, the base registers are malicious (and therefore server S), as shown in the following example: Let o_w and o'_w be two sequential WRITE operations of C_i , o'_w precedes READ operation o_r of C_j but it is hidden by the malicious base registers such that o_r sees only o_w . As o'_w precedes o_r , o'_w cannot see o_r . However, as o_r sees o_w , it expects that o'_w will see o_r . The next WRITE operation o''_w of C_i will write this information. If client C_j sees o''_w , which would violate weak fork-linearizability, the check, explained above, is not passed.

4.5.2 Description of the WFL Protocol

This section explains the algorithm of client C_i to implement the high-level READ and WRITE operations of the WFL protocol. The WFL protocol is given as Algorithm 4.4, the used variables in Algorithm 4.3.

Algorithm 4.3: Variables used in Algorithm 4.4 by Client C_i

Variables:

S , atomic snapshot object with n components, initially $((0, \perp), \dots, (0, \perp))$
/* timestamp+sig */
 W_1, \dots, W_n , SWMR atomic registers, initially $(\perp, 0, \emptyset, \emptyset, \perp)$
/* val+ts+rs+ws+sig */
 v, wv value, initially \perp /* value written to storage */
 $wts, ots, i, k, r, r', w, w', tmp_1, \dots, tmp_n$ integer, initially 0
/* timestamps + temp. variables */
 $read_seen[1..n][1..n], write_seen[1..n][1..n]$,
 $r_write_seen[1..n][1..n]$, matrix of sets of pairs (integer, integer), initially \emptyset
/* matrices of seen operations */
 sig, sig_1, \dots, sig_n signature, initially \perp /* signatures */

At invocation of high-level READ(j), client C_i increments its local timestamp and generates a digital signature of it. The signed timestamp is stored to snapshot object SO using operation UPDATE($(ots, sig), SO, i$) (lines 4.4.2–4.4.4). Then, client C_i reads register W_j and verifies the signature (lines 4.4.5–4.4.6). The content of register W_j contains the written value wv , the corresponding timestamp wts , as well as two matrices r_read_seen and r_write_seen . Both matrices are of size $n \times n$ where each entry holds a set of integer pairs (r, w) . Client C_i maintains a variable $read_seen$ of the same type, where a pair $(r, w) \in read_seen[i][j]$ denotes that READ of client C_i with timestamp r has seen WRITE of client C_j with timestamp w . Analogously, client C_i maintains a second matrix $write_seen$, where $(r, w) \in write_seen[i][j]$ denotes that WRITE of client C_i with timestamp w has seen READ of client C_j with timestamp r . In the next step (line 4.4.7), client

Algorithm 4.4: WFL Protocol, Algorithm of Client C_i

```

4.4.1 READ( $j$ ) do
4.4.2    $ots \leftarrow ots + 1$                                 /* increment timestamp */
4.4.3    $sig \leftarrow \text{sign}_i(ots)$                         /* signature on timestamp */
4.4.4   UPDATE( $((ots, sig), S, i)$ )                        /* update call to snapshot object */
4.4.5    $(wv, wts, r\_read\_seen, r\_write\_seen, sig) \leftarrow \text{read}(W_j)$  /* low-level read */
4.4.6   if not verify $_j(sig)$  then halt                  /* signature verified? */
4.4.7    $read\_seen \leftarrow \text{merge}(read\_seen, r\_read\_seen)$  /* update read\_seen */
4.4.8    $read\_seen[i][j] \leftarrow read\_seen[i][j].\text{add}((ots, wts))$  /* add seen write */
4.4.9   check()                                           /* check passed? */
4.4.10   $write\_seen \leftarrow \text{merge}(write\_seen, r\_write\_seen)$  /* update write\_seen */
4.4.11  return  $wv$                                        /* return read value */

4.4.12 WRITE( $i, v$ ) do
4.4.13   $ots \leftarrow ots + 1$                                 /* increment timestamp */
4.4.14   $sig \leftarrow \text{sign}_i(v, ots, read\_seen, write\_seen)$  /* signature on timestamp */
4.4.15   $\text{write}((v, ots, read\_seen, write\_seen, sig), W_i)$  /* low-level write */
4.4.16   $((tmp_1, sig_1), \dots, (tmp_n, sig_n)) \leftarrow \text{SCAN}(S)$  /* scan call to S0 */
4.4.17  for  $k = 1, \dots, n$  do
4.4.18    if not verify $_k(sig_k)$  then halt                /* signature verified? */
4.4.19     $write\_seen[i][k] \leftarrow write\_seen[i][k].\text{add}((tmp_k, ots))$  /* add all seen reads */
4.4.20  return OK                                       /* successfully return */

4.4.21 check() do
4.4.22  for  $k = 1, \dots, n$  do
4.4.23    forall  $(r, w) \in read\_seen[k][i]$  do
      /* check if own writes have seen read operations reading
      my values */
4.4.24    if  $\exists(r', w') \in write\_seen[i][k]$  s.t.  $w' > w$  and  $w'$  minimal then
4.4.25      if  $r' < r$  then halt
4.4.26    forall  $(r, w) \in read\_seen[i][k]$  do
      /* check if own reads have been seen by other's write
      operations */
4.4.27    if  $\exists(r', w') \in r\_write\_seen[k][i]$  s.t.  $w' > w$  and  $w'$  minimal then
4.4.28      if  $r' < r$  then halt

```

C_i “merges” variables r_read_seen and $read_seen$. The merge procedure returns for each entry of two $n \times n$ set matrices A, B set $A[i][j] \cup B[i][j]$, $i, j = 1, \dots, n$. Then, C_i adds a pair consisting of its current timestamp and timestamp wts from W_j to $read_seen[i][j]$. To ensure weak fork-linearizability, client C_i calls procedure “check” (line 4.4.9). If all checks are passed, C_i merges r_write_seen and $write_seen$ and returns value wv (lines 4.4.10–4.4.11).

At invocation of WRITE(i, v), client C_i increments its timestamp (line 4.4.13). It

digitally signs value v , its timestamp, and variables $read_seen$ and $write_seen$ to write to register W_i (lines 4.4.14–4.4.15). Next, it reads all timestamps of READS by calling SCAN to snapshot object SO (line 4.4.16). All entries in SO are digitally signed and thus client C_i verifies the signatures (line 4.4.18). Then, it adds to all sets $write_seen[i][k]$ ($k = 1, \dots, n$) a pair consisting of the timestamp of the k th component of SO and C_i 's current timestamp (line 4.4.19). Finally, client C_i successfully returns (line 4.4.20).

Procedure “check” implements the principle sketched in section 4.5.1 for n clients. It ensures that *weak fork-linearizability* is never violated. The procedure, called by C_i during $READ(j)$ (line 4.4.21), moves through a loop performing two checks: The first check (line 4.4.24–4.4.25) considers the information left by clients during $READ(i)$ operations (this information is stored in the i th column of $read_seen$). If $READ(i)$ with timestamp r of client C_k has seen WRITE of C_i with timestamp w , then it is tested whether the next WRITE of C_i has read (using SCAN) timestamp r or higher of client C_k . The check uses the local $write_seen$ variable of C_i . The second check (line 4.4.27–4.4.28) reviews the information left by client C_i during any $READ(k)$ (which is kept in the i th row of $read_seen$). If $READ(k)$ with timestamp r of client C_i has seen WRITE of C_k with timestamp w , then we check whether the next WRITE of C_k has read (using SCAN) timestamp r or higher of client C_i . This check requires matrix $r.write_seen$, which has been fetched from W_j in line 4.4.5 before procedure “check” is called.

4.5.3 Correctness Arguments

In this section we give the intuition why the WFL protocol in Algorithm 4.4 satisfies the properties of a wait-free Byzantine emulation of a shared memory with weak fork-linearizability. Intuitively, the definition of weak fork-linearizability requires for each client C_i to construct a sequence π_i such that causality among operations, the sequential specification of a shared memory, and weak real-time order is satisfied, and that two sequences π_i and π_j share the same prefix up to the second last common operation (at-most-one-join). The proof proceeds in steps, where in the first step all operations that have to be included in sequence π_i are causally ordered. Next, this order is extended such that it additionally respects the sequential specification. Intuitively, as all written values are digitally signed, the sequential specification never interferes with causality. The hardest step is to prove, that this order can be further refined such that it does not violate the weak real-time order. The intuition for this is given below as a proof by contradiction:

We assume that $READ(j)$ operation o_r of client C_i does not return the latest value v' , written by $WRITE(j, v')$ operation o'_w , but an older value written by operation o_w (see Figure 4.4). Further, let o_r be not the last operation of C_i

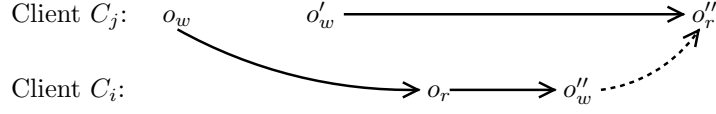


Figure 4.4: Correctness Idea of the WFL Protocol in Algorithm 4.4. Arrows denote the “seen” relation.

during the execution of the WFL protocol. During operation o_r , the pair (r, w) ⁶ is added to set $read_seen[i][j]$. The data written by the next $WRITE(i, v'')$ operation o''_w of C_i contains this information. Now, the algorithm prevents client C_j from reading the value v'' written by o''_w which would violate weak real-time order (as o_r is ordered before o'_w according to the sequential specification). When during o''_r C_j sees operation o''_w , it finds the pair (r, w) in r_read_seen . As o'_w precedes o_r , it could not have seen o_r , thus $write_seen[j][i]$ contains a pair (r', w') such that $r' < r$ and the check in line 4.4.25 is not passed. Hence, operation o''_r of client C_j would block — a contradiction. This implies that such a situation does not appear and the constructed order of operations also satisfies weak real-time order.

As the last step, showing that the sequences π_i satisfy the at-most-one-join property follows directly from a simple construction argument. To prove liveness, as required in the definition of a Byzantine emulation (Definition 2.4 on page 30), we show that no operation blocks when all base objects are correct, which follows from the principle sketched in section 4.5.1 as in this case all checks are passed.

4.5.4 Proof of Correctness of the WFL Protocol

Finally, this section proves that the WFL protocol (given as Algorithm 4.4) implements a wait-free, weakly fork-linearizable Byzantine emulation of a shared memory. The properties of weak fork-linearizability have been given in Definition 4.4 on page 69.

To show the existence of sequential permutations π_i for every client C_i that satisfy weak fork-linearizability, let o_{i1}, \dots, op_{il_i} be the operations of C_i ordered by their timestamps, $i \in 1, \dots, n$, $l_i \in \mathbb{N}$. The timestamp of an operation is represented by variable ots that is assigned to a $READ$ operation in line 4.4.4 and to a $WRITE$ operation in line 4.4.15, respectively. For every client C_i we define a directed graph G_{π_i} , where the set of operations o_{i1}, \dots, op_{il_i} are the vertices. For all $k \in 1, \dots, l_i - 1$, we draw an edge from o_{ik} to $o_{i(k+1)}$.

Next, we construct a directed graph G_π as $G_{\pi_1} \cup G_{\pi_2} \cup \dots \cup G_{\pi_n}$. We add an edge from o_{iw_i} to o_{jr_j} to graph G_π if o_{iw_i} is a $WRITE(i, v)$ operation of C_i , $i \in 1, \dots, n$, and o_{jr_j} is a $READ(j)$ operation of C_j , $j \in 1, \dots, n$, that reads value v written by

⁶We assume that operation o_x is assigned timestamp x .

o_{iw_i} .

The purpose of the next Corollary is to show that a partial order of the operations can be defined according to the ordering of the vertices of graph G_π .

Corollary 4.16. Graph G_π does not contain directed cycles.

Proof. Let us assume that there exists the following directed cycle which is also the shortest possible one: $(o_{ir}, o_{iw}, o_{jr}, o_{jw})$, where o_{ir} is a READ and o_{iw} is a WRITE operation of client C_i , and o_{jr} is a READ and o_{jw} is a WRITE operation of client C_j . Further, let o_{ir} have a lower timestamp than o_{iw} , let o_{jr} read the value written by o_{iw} , let o_{jr} have a lower timestamp than o_{jw} , and let o_{ir} read the value written by o_{jw} . We now can deduce the following statements:

1. o_{ir} precedes o_{iw} , as both are operations of C_i and o_{ir} has a lower timestamp than o_{iw} (line 4.4.13).
2. o_{jr} completes after o_{iw} has been invoked, as otherwise o_{jr} cannot read the value written by o_{iw} (written values are digitally signed).
3. o_{jr} precedes o_{jw} , as both are operations of C_j and o_{jr} has a lower timestamp than o_{jw} (line 4.4.13).
4. o_{jw} is invoked after o_{iw} has been invoked, by 2. and 3.
5. o_{ir} completes after o_{jw} has been invoked, as otherwise o_{ir} cannot read the value written by o_{jw} (written values are digitally signed).

Statements 1.–5. lead to a contradiction as operation o_{ir} may not at the same time complete before *and* after o_{iw} is invoked. Analogous arguments hold, if the circle is extended between o_{jw} and o_{ir} to a circle $(o_{ir}, o_{iw}, o_{jr}, o_{jw}, \dots)$ of arbitrary length. Hence, graph G_π does not contain directed cycles. \square

For each client C_i we recursively define the subgraph $T(o_{il_i})$ that contains o_{il_i} as a vertex, and if o is a vertex of $T(o_{il_i})$, and (o', o) is an edge of G_π , then vertex o' and edge (o', o) is added to $T(o_{il_i})$ until no more edges can be added.

Corollary 4.17. The set of operations represented by the vertices of $T(o_{il_i})$ contains all operations of client C_i .

Proof. By construction of graph G_{π_i} , there is a path from any operation of client C_i to o_{il_i} . Thus, all operation of C_i are contained in $T(o_{il_i})$. \square

Now, we start constructing for each client C_i a subsequence π_i of the history σ of any execution of the WFL protocol that satisfies the properties of weak fork-linearizability (Definition 4.4). The next corollary constructs an order relation among operations in π_i .

Corollary 4.18. There is a sequential permutation π_i of the set of operations represented by the vertices of $T(o_{il_i})$ and an order relation $<_{\pi_i}$ that satisfies the following condition: For every operation $o \in \pi_i$ and every WRITE operation $o' \in V(G_\pi)$ s.t. o' causally precedes o , it holds that $o' \in \pi_i$ and that $o' <_{\pi_i} o$.

Proof. By Corollary 4.17, we know that every operation of C_i is contained in π_i . Further, by the construction of graphs $G(\pi)$ and $T(o_{il_i})$ every operation that causally precedes an operation in π_i (Definition 4.1) is contained in π_i . As $T(o_{il_i})$ contains no cycles (Corollary 4.16), for $o, o' \in \pi_i$ we order o before o' ($o <_{\pi_i} o'$) if there is an edge from o to o' in $T(o_{il_i})$. \square

The order relation constructed in the proof of Corollary 4.18 does not necessarily respect the sequential specification of a shared memory. The next corollary shows how this can be achieved.

Corollary 4.19. The order relation $<_{\pi_i}$, constructed in Corollary 4.18 can be extended such that π_i satisfies the sequential specification of a shared memory: If o_{kr} is a READ operation of some client C_k , $k \in 1, \dots, n$, in π_i that reads the value written by WRITE operation o_{lw} of client C_l , $l \in 1, \dots, n$, then we additionally order o_{kr} before $o_{l(w+1)}$ where $o_{l(w+1)}$ is the next WRITE operation of C_l in π_i (if it exists in π_i).

Proof. By causality o_{lw} is ordered before o_{kr} . There is no WRITE operation of C_l between o_{lw} and o_{kr} in π_i , as $o_{l(w+1)}$ is the next WRITE operation of C_l in π_i after o_{lw} , and o_{kr} can be ordered before it. \square

Now we define how the remaining operations have to be ordered such that π_i satisfies weak real-time ordering. The proof of the following lemma distinguishes two cases to show that when a READ reads a value written by some WRITE, then the WRITE is the last one that precedes the READ. The two cases correspond to the fact that a READ operation of client C_i may appear in its own sequence π_i (case B) as well as in sequence π_j of C_j .

Corollary 4.20. The order relation $<_{\pi_i}$, constructed in Corollaries 4.18 and 4.19 does not violate weak real-time order.

Proof. We distinguish the following cases:

Case A: Let w and w' be two WRITE operations of client C_i and let r be a READ operation of client C_j that reads the value written by w . Let w precede w' in π_i . Let r be not the last operation of C_j in π_i . Then w' does not precede r , i.e., $r.UPDATE$ precedes $w'.SCAN$ ⁷.

⁷In the following $x.UPDATE$ ($x.SCAN$) denotes a call of procedure `UPDATE` (`SCAN`) during `READ` (`WRITE`) operation x in line 4.4.4 (line 4.4.16). The analogous notation holds for $x.write$ ($x.read$) in line 4.4.5 (line 4.4.15) during Lemma 4.25.

Proof. We assume by contradiction that WRITE operation w' precedes READ operation r , i.e., $w'.\text{SCAN}$ precedes $r.\text{UPDATE}$ (Ass. A). By assumption, operation r reads the value written by operation w and thus by line 4.4.8 set $\text{read_seen}_j[j][i]$ at client C_j contains the pair (r, w) ⁸. Let $w_{\min} \leq w'$ be the WRITE operations of client C_i directly following w . Then, during operation w_{\min} , by line 4.4.17, the pair (x_{\min}, w_{\min}) is added to set $\text{write_seen}_i[i][j]$ at client C_i . By Ass. A and as all written timestamps are digitally signed, it holds that $x_{\min} < r$. As r is not the last operation of C_j in π_i , there exists a read operation r'' of C_i that succeeds w' , a write operation w'' of C_j that succeeds r , and there exists a path in graph $T(\text{op}_{il_i})$ from r to r'' that contains w'' (see Figure 4.5). During operation w'' of client C_j , variable read_seen_j is written by line 4.4.15. As there is the causal path from w'' to r'' , by lines 4.4.7 and 4.4.15, during operation r'' of client C_i , $r.\text{read_seen}[j][i]$ contains pair (r, w) . By line 4.4.7 it is also contained in $\text{read_seen}_i[j][i]$ during operation r'' . We further know that $\text{write_seen}_i[i][j]$ at client C_i contains (x_{\min}, w_{\min}) . As operation w_{\min} is the minimal operation larger than w , the check in line 4.4.25 is not passed as $x_{\min} < r$ and operation r'' blocks. This means that r is not in π_i — a contradiction. \square

Case B: Let w and w' be two WRITE operations of client C_j and let r be a READ operation of client C_i that reads the value written by w . Let w precede w' in π_i . Let w' be not the last operation of C_j in π_i . Then w' does not precede r , i.e., $r.\text{UPDATE}$ precedes $w'.\text{SCAN}$.

Proof. We assume by contradiction that WRITE operation w' precedes READ operation r , i.e., $w'.\text{SCAN}$ precedes $r.\text{UPDATE}$ (Ass. B). By assumption, operation r reads the value written by operation w and thus by line 4.4.8 set $\text{read_seen}[i][j]_i$ at client C_i contains the pair (r, w) . Let $w_{\min} \leq w'$ be the WRITE operations of client C_j directly following w . Then, during operation w_{\min} , by line 4.4.17, the pair (x_{\min}, w_{\min}) is added to set $\text{write_seen}_j[j][i]$ at client C_j . By Ass. B and as all written timestamps are digitally signed, it holds that $x_{\min} < r$. As w' is not the last operation of C_j in π_i , there exists a read operation r'' of C_i that succeeds r , a write operation w'' of C_j that succeeds w' , and there exists a path in graph $T(\text{op}_{il_i})$ from w to r'' that contains w' and w'' (see Figure 4.5). During operation w'' of client C_j , variable write_seen_j is written by line 4.4.15. As there is the causal path from w'' to r'' , by lines 4.4.10 and 4.4.15, during operation r'' of client C_i , $r.\text{write_seen}[j][i]$ contains pair (x_{\min}, w_{\min}) . We further know

⁸To simplify the presentation, let r denote the timestamp assigned to operation r and w the timestamp assigned to w .

that $read_seen_i[i][j]$ at client C_i contains (r, w) . As operation w_{\min} is the minimal operation larger than w , the check in line 4.4.28 is not passed as $x_{\min} < r$ and operation r'' blocks. This means that w' is not in π_i — a contradiction. \square

It remains to show that in π_i READ operations of client C_j that read values written by operations of client C_k can be ordered to satisfy weak real-time order. The proof is obvious as weak real-time order holds for π_j when case B from above is applied.

Hence, the order induced in Corollary 4.19 does not violate weak real-time order — i.e., the not yet ordered operations can be ordered in real-time order or in any deterministic order if they are concurrent. \square

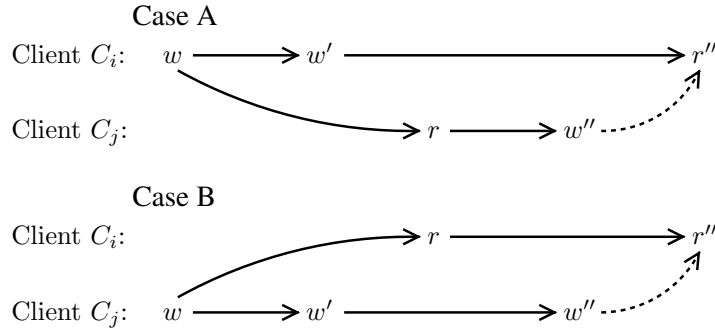


Figure 4.5: Proof of Corollary 4.20. Arrows denote the causality relation.

Corollary 4.21. If all operations in π_i which have not yet been ordered in Corollary 4.18 or 4.19 are ordered according to their real-time order if they are sequential and by the real-time order of their completion event else, then order relation $<_{\pi_i}$ of π_i satisfies weak real-time ordering.

Proof. The proof follows directly from construction and Corollary 4.20. \square

To complete the correctness proof, we have to add operations to each π_i such that the join-at-most-once property is satisfied. This is because π_i may contain operations of π_j but none of π_k , but π_j might have common operations with π_k . To ensure that π_i and π_j share a common prefix such operations have to be added to π_i . Thus, we define a merge operation on totally ordered command sequences π_i .

Definition 4.22. Let π_i and π_j be two totally ordered command sequences such that there are at least two operation o, o' for which holds $o \in \pi_i \cap \pi_j$ and $o' \in \pi_i \cap \pi_j$ ⁹.

⁹By construction, for all such operations hold $o <_{\pi_i} o'$ and $o <_{\pi_j} o'$ or $o >_{\pi_i} o'$ and $o >_{\pi_j} o'$.

Let $\pi|_x$ denote the prefix of an operation sequence π that ends with operation x . To *merge* π_i and π_j we perform the following steps: Let $o_{2\text{ndlast}}$ be the second last operation in $\pi_i \cap \pi_j$. In π_i and π_j we replace the prefix $\pi_i|_{o_{2\text{ndlast}}}$ and $\pi_j|_{o_{2\text{ndlast}}}$ by $\pi_{\text{merge}_{ij}}$:

- $\pi_{\text{merge}_{ij}}$ contains all operations from $\pi_i|_{o_{2\text{ndlast}}} \cup \pi_j|_{o_{2\text{ndlast}}}$
- If for two operations o, o' in $\pi_{\text{merge}_{ij}}$ holds $o <_{\pi_i} o'$ or $o <_{\pi_j} o'$, then we order o before o' in $\pi_{\text{merge}_{ij}}$, i.e., $o <_{\pi_{\text{merge}_{ij}}} o'$.
- If for two operations o, o' in $\pi_{\text{merge}_{ij}}$ neither $o <_{\pi_{\text{merge}_{ij}}} o'$ nor $o >_{\pi_{\text{merge}_{ij}}} o'$ holds, then we order o and o' in $\pi_{\text{merge}_{ij}}$ according to their real-time ordering or by the real-time order of their completion event if they are concurrent.

Corollary 4.23. For all pairs, $i, j \in 1, \dots, n$, if we *merge* π_i and π_j whenever they have two or more operations in common until no more changes appear. Then sequences π_i, \dots, π_n satisfy the *at-most-one-join* property (Definition 4.4).

Proof. Correctness follows directly from the construction given in Definition 4.22. \square

Lemma 4.24. The history σ of an execution of the WFL protocol satisfies weak fork-linearizability with respect to a shared memory for n clients.

Proof. The correctness follows from Corollaries 4.17 and 4.19 which ensures that π_i is a view of σ with respect to the functionality of a shared memory, from Corollary 4.18 that guarantees that causality is respected, from Corollary 4.21 that ensures weak real-time ordering, and Corollary 4.23 that guarantees the *at-most-one-join* property. \square

Lemma 4.25. If server S implementing the base registers and the snapshot object SO is correct, and σ is the history of an execution of the WFL protocol, then no READ operation blocks in line 4.4.6, 4.4.25, nor 4.4.28 and no WRITE operation blocks in line 4.4.18.

Proof. We show that no operation of the WFL protocol blocks when the base registers behave correctly. As the clients behave correctly and registers do not forge signatures, it is easy to see that WRITE operations do not block. The same argument holds for the check in line 4.4.6 during READ operations. Thus, it remains to show that READ operations do not block in line 4.4.25 and 4.4.28.

Let us assume for contradiction that there is a READ operation of client C_i that blocks in line 4.4.25 or 4.4.28:

Line 4.4.25 There exists a READ operation r of client C_k that has read from WRITE operation w of client C_i (line 4.4.8). By assumption, the minimal WRITE operation of C_i after w , called w' has seen READ operation r' of client C_k (line 4.4.17). As $r' < r$, and as all registers are correct, $r.UPDATE$ does not precede $w'.SCAN$. Thus, $w'.write$ precedes $r.read$. However, as w precedes w' , we conclude that r reads the value written by w' — a contradiction.

Line 4.4.28 There exists a READ operation r of client C_i that has read from WRITE operation w of client C_k (line 4.4.8). By assumption, the minimal WRITE operation of C_k after w , called w' has seen READ operation r' of client C_i (line 4.4.17). As $r' < r$, and as all registers are correct, $r.UPDATE$ does not precede $w'.SCAN$. Thus, $w'.write$ precedes $r.read$. However, as w precedes w' , we conclude that r reads the value written by w' — a contradiction.

Hence, no operation in σ blocks. □

Finally, it has been shown in Lemma 4.24 (Weak Fork-Linearizability), and Lemma 4.25 (No Blocking), that the WFL protocol in Algorithm 4.4 correctly implements a wait-free, weak fork-linearizable Byzantine emulation of a shared memory.

4.6 Analysis & Conclusion

The AFL protocol given as Algorithm 4.1 requires server S to provide n atomic registers plus n additional ones to implement the INC&READ counter. The operations of the AFL protocol have an overall communication complexity of $O(n^2)$, as the size of the version vectors used in Algorithm 4.1 is linear in the number of clients n and as a linear number of such version vectors are exchanged per operation. In contrast, the *lock-step* protocol of Cachin *et al.* [CSS07], also based on linear size version vectors, has an overall communication complexity of $O(n)$. This difference results from the fact that the server used by Cachin *et al.* is computationally strong enough to select the latest written version vector while in Algorithm 4.1 the client is required to read from *all* registers on server S to find the latest one by itself. For the WFL protocol given as Algorithm 4.4, n atomic registers plus $2n$ additional ones for the atomic snapshot object have to be implemented by server S . The WFL protocol, uses matrices of size $n \times n$ where the size of each entry depends on the total number of operations N , resulting in a communication complexity of $O(N \cdot n^2)$. The thesis at hand leaves as an open problem whether this complexity can be reduced by implementing a “garbage collection”. However, both the AFL and the WFL protocol require only a linear number of base registers on server S .

The AFL and the WFL protocol demonstrate as a first known result that fork-consistent semantics can be implemented on a server only providing registers.

The AFL protocol satisfies fork-linearizability and implements a shared functionality of universal type. Similar to non-fork-consistent universal constructions from registers, the AFL protocol may abort operations under concurrency. Hence, fork-linearizability may be “added” to such protocols without making additional assumptions. The WFL protocol implements a shared memory that ensures *weak* fork-linearizability and where operations are wait-free as long as server S implementing the base registers behaves correctly. Weak fork-linearizability is the strongest known fork-consistency property that may be implemented in a wait-free manner. Although it weakens fork-linearizability, it has shown to be of practical relevance [CKS11]. Moreover, the WFL protocol shows for the first time that registers are sufficient to implement a fork-consistent shared memory. So far, all existing implementations are based on computationally stronger objects (featuring read-modify-write operations [KRS88]). This thesis leaves as an open question whether there is a weak fork-linearizable emulation of a universal type providing a stronger liveness condition than *abortable* in the fault-free case.

The AFL and WFL protocols demonstrate that for a broad range of fork-consistent implementations simple memory objects are sufficient. Besides the relevance for the theoretical research community, this result matters also for practical systems. Recent cloud service providers offer different interfaces from virtual machines (e.g., Amazon EC2) to read/write APIs in the form of e.g., key-value-stores (Amazon S3). Typically the latter is much less expensive which makes the AFL and WFL protocols preferred approaches also with respect to monetary aspects.

5 Lightweight Atomic Storage

This chapter introduces two Byzantine fault-tolerant distributed protocols constituting contributions (C5) and (C6) of the thesis at hand. The chapter tackles the problem how to implement fault tolerant shared storage with atomic operations using $3t + 1$ servers out of which t may be Byzantine faulty. The focus is on *lightweight* protocols that do not rely on data authentication, require the minimal number of replicated servers, and exhibit optimal worst-case latency for READ and WRITE operations. The LWR protocol (contribution (C5)) is a Byzantine fault-tolerant implementation of an atomic register. Protocol LwKVS (contribution (C6)) extends the atomic register implemented by the LWR protocol and provides the functionality of a key-value store. At the core of both protocols a *commitment scheme* is employed to improve on existing protocols by speeding up reading and supporting an unbounded number of (possibly) malicious readers. Since the LWR and LwKVS protocols do not rely on expensive inter-server communication, replicas can be spread across multiple providers for failure independence at no overhead.

In the past decades the research area of fault-tolerant storage implementations has been extensively explored. The published results indicate a fundamental difference between implementations that use self-verifying data (e.g., digital signatures) and those that do not rely on such data authentication. This difference manifests in the latency of the implemented operations measured in the number of communication rounds. In the authenticated data model, WRITE operations of a Byzantine fault-tolerant atomic register can be implemented that achieve a latency of a single round, while in the non-authenticated data model two communication rounds are required. An even larger gap exists for atomic READ operations: a two-round latency for authenticated data versus four communication rounds for atomic reads with non-authenticated data. Hence, system developers facing a trade-off between two unfavorable options: Taking the performance and administrative overhead induced by data authentication or accepting the suboptimal (read) latency in the non-authenticated data model. With respect to this issue the LWR protocol constitutes an attractive and *lightweight* solution: It does not require self-verifying data but is able to implement atomic READ and WRITE operations that achieve latencies close to the fastest implementations in the authenticated data model: a two-round latency for both READ and WRITE operations. Moreover, by employing a commitment scheme it constitutes the first atomic register implementation that

does not require authenticated data and allows an unbounded number of possibly malicious readers. The introduced LwKVS protocol maintains all the desirable properties of the LWR protocol in the context of a key-value-store, that has become the most applied storage abstraction for recent services in the cloud.

Section 5.1 gives background information about the two main abstractions *atomic register* and *key-value-store* explored in this chapter. Related works are studied in Section 5.2. Section 5.3 gives a refinement of the system model of Chapter 2. The LWR and LwKVS protocols are given as contribution (C5) in Section 5.4 and as contribution (C6) in Section 5.5. Section 5.6 discusses the introduced protocols and further optimizations for future research.

5.1 Introduction

5.1.1 Atomic Registers

In Chapter 4, the fork-consistent emulations of a universal object and a shared memory are based on *atomic registers*. In the previous chapter, these atomic registers have been regarded as being untrusted, i.e., they may deviate in an arbitrary way from their specified behavior. However, to achieve a maximal benefit for the user, the atomic registers are expected to behave as specified for most of the time. The malicious behavior should be the exception. Hence, an obvious challenge is to implement the atomic registers themselves in a fault-tolerant manner to mitigate the negative influence of maliciously behaving system components. The key goal of this chapter is to introduce a Byzantine fault-tolerant implementation of an atomic register from a set of replicated servers. Thereby, a fraction of the replicated servers is allowed to exhibit Byzantine faulty behavior. Replication allows to *mask* [AS85] malicious behavior such that the implemented atomic register is *safe* and *live* even in the presence of Byzantine faults.

For practical settings it is of vital importance to keep the costs incurred by replication minimal — i.e., for a given number t of malicious replicas to find the required total number n of replicated servers, which is also denoted as *optimal resilience*. Such a replicated implementation is denoted as *robust* [ABND95] if it is optimally resilient and provides wait-free operations. It has been shown, that $n = 3t + 1$ replicas are required to tolerate t Byzantine faulty ones [MAD02]. Besides low replication costs, for practical settings further aspects are desirable to achieve a *lightweight* implementation: to attain low latency, the number of communication rounds between the clients and the servers has to be minimal. To avoid the costs related with public-key cryptography, an implementation not relying on self-verifying data is desirable. For reasons of scalability, an unbounded number of readers should be supported out of which any number may be malicious.

Although fault-tolerant registers have been intensively explored by the research

community in the past decades, none of the existing solutions is able to satisfy all above mentioned requirements. Recent work [ACKM06] proves that any Byzantine-tolerant storage employing at most $4t$ replicated servers has at least some write operation completing in two communications rounds. Work of Guerraoui and Vukolic [GV06] rules out reading in a single-round even from robust *safe* registers. By allowing the existence of *secret tokens*, Dobre *et al.* [DMSS09] circumvent this impossibility result only for robust *regular* storage. Finally, any robust atomic register implementation where READ operations complete in two communication rounds has shown to be optimal, even for the stronger model of authenticated data [DGLV10]. WRITE operations that complete after two communication rounds are optimal for the unauthenticated data model [ACKM06].

This chapter introduces a lightweight, Byzantine fault-tolerant, robust implementation of a single-writer multiple-reader (SWMR) atomic register (LwR protocol). The LwR protocol is denoted as *lightweight*, as it satisfies optimal resilience, features an optimal latency of two communication rounds for READ and WRITE operations, does not rely on self-verifying data, is scalable in the number of clients as the communication complexity is in $\mathcal{O}(n)$, and tolerates any number of malicious readers.

The basic principle underlying the implemented atomic register is that it makes use of a cryptographic commitment scheme [BCC88]. During the pre-write phase¹ of a WRITE(v) operation, the writer generates a secret token and sends, along with value v , a *commitment* to the servers. By the *hiding* property of the commitment scheme, the chosen token cannot be known by the servers. In the write-phase², the writer sends an *opening* to the servers, revealing the token. The *binding* property of the commitment ensures that the servers can validate the revealed token. The employed commitment scheme is given in two implementation variants (H) and (S). Variant (H) is based on a collision-resistant hash function, while variant (S) is based on Shamir's secret sharing scheme [Sha79].

Accessing $t + 1$ servers that report a validation of the token of a candidate value v indicates to a client during a READ() operation that the corresponding pre-write phase is complete. Thus, value v has been stored to $t + 1$ *correct* servers. This is a sufficient condition to perform the write-back of v , to ensure atomic semantics, in a single communication round.

The introduced LwR protocol implementing a SWMR atomic register constitutes the first robust and latency-optimal implementation in the unauthenticated data model that tolerates an unbounded number of possibly malicious readers. It is assumed that malicious servers may not predict the output of a (pseudo) random number generator [DMSS09] at the writer.

¹The first communication round during a WRITE operation.

²The second communication round during a WRITE operation.

5.1.2 Key-Value Store

In the last decade, we have observed an increasing trend of storage solutions provided by “cloud services”. A key advantage of storing data in the cloud is for the user that data is always accessible. To address the user needs for high reliability and availability, cloud service providers rely on replication techniques. Surprisingly, simple read/write interfaces for cloud storage solutions have become popular. A key-value store (KVS) is the most adopted storage abstraction for cloud services providing such a simple interface [DHJ⁺07, MTJ⁺08, ALM⁺10, LM10, CWO⁺11]. A KVS allows concurrent access of several clients to store and retrieve data in the cloud. It usually provides four different operations to the clients [BCE⁺12] (also see Section 2.1.3 in Chapter 2): a $\text{PUT}(key, v)$ operation stores value v under a unique key key at the KVS. Operation $\text{GET}(key)$ retrieves the correct value associated with key from the KVS. The $\text{DELETE}(key)$ operation removes the value associated with key , while operation $\text{LIST}()$ returns a list of all keys with associated values.

Obviously, the interface of an atomic register and a KVS are very similar: PUT operations of a KVS correspond to WRITE operations of an atomic register, while GET corresponds to READ operations. As for the same practical reasons lightweight solutions are desired for KVSs, the LwKVS protocol as contribution (C6) of this thesis extends the SWMR atomic register implemented by the LwR protocol to a Byzantine fault-tolerant *key-value store*. As well, the proposed KVS is optimally-resilient with respect to Byzantine faulty servers, supports an unbounded number of clients, and tolerates an unbounded number of malicious readers. Operations $\text{GET}(key)$ complete in two communication rounds. As the KVS implemented by the LwKVS protocol supports multiple writers an extra round of communication is required. Thus, $\text{PUT}(key, v)$ and $\text{DELETE}(key)$ operations complete after three rounds of communication. Operations, $\text{GET}(key)$, $\text{PUT}(key, v)$, and $\text{DELETE}(key)$ are atomic, while $\text{LIST}()$ operations feature *regular* semantics.

Contributions of Chapter 5 Specifically, this chapter makes the following contributions:

- The LwR protocol, a robust, Byzantine fault-tolerant implementation of a single-writer multiple-reader *atomic register* as contribution (C5) of this thesis. The atomic register implemented by the LwR protocol is called *lightweight*, as it is optimal in the number of replicated servers $n = 3t + 1$ to tolerate t Byzantine faulty servers, and in the number of communication rounds which is two rounds for each READ and WRITE operation. Further, the LwR protocol does not rely on self-verifying data, is scalable in the

number of clients as the communication complexity is in $\mathcal{O}(n)$, and tolerates any number of malicious readers.

- The LwKVS protocol, a robust, lightweight Byzantine fault-tolerant implementation of a key-value store as contribution (C6) of this thesis. The LwKVS protocol is optimally resilient and provides wait-free operations. Operations PUT and DELETE require three and operations GET and LIST require two communication rounds. As the LwR protocol, it does not rely on self-verifying data, is scalable in the number of clients and tolerates any number of malicious readers.

Note that both protocols LwR and LwKVS make use of a *commitment scheme*. This chapter introduces two variants (H) and (S) to implement the commitment scheme. Variant (H) employs a collision-resistant hash function and variant (S) makes use of a secret sharing scheme as introduced by Shamir [Sha79].

5.2 Related Work

Fault-tolerant Storage The question how to implement fault-tolerant storage has been extensively studied over the past decades. Lamport introduced the notions of *safe*, *regular*, and *atomic* registers [Lam86] as well-known and widely accepted abstractions for shared storage. *Robust* [ABND95] implementations are of particular interest, as they are optimally resilient and provide wait-free [Her91] operations. In a model where faulty servers are only allowed to crash, Attiya *et al.* [ABND95] introduced in a seminal work an implementation of a robust, atomic register where WRITE operations finish after one and READ operations after two communication rounds with the clients. Note, that for the crash-fault model, an implementation tolerating t out of $2t + 1$ faulty servers is denoted as *optimally resilient*. Reducing the communication complexity to a single rounds for READ operations comes at the cost of more servers implying a non-optimal resilience [DGLC04]. This shows also fundamental differences between fault-tolerant implementations of *regular* and *atomic* registers: For a robust *regular* register, single-round READ and WRITE operations are possible [ABND95], while this is not true for robust *atomic* register implementations, where readers even have to modify [DGLC04] the state of $t + 1$ servers [FL03].

In the Byzantine failure model, it has been shown, that $n = 3t + 1$ servers are required to tolerate t Byzantine faulty ones [MAD02], defining optimal resilience in such a setting. Work of Abraham *et al.* [ACKM06] ruled out the the existence of robust Byzantine fault-tolerant register implementation where all WRITE operations finish after a single communication round. The exact conditions that enable single-round atomic READ operations are given in an extended work of

Dutta *et al.* [DGLV05]. If readers are not allowed to write, $t + 1$ communication rounds constitute a tight lower bound for READ operation, even for weaker *safe* registers [ACKM06]. If readers may modify the state of the servers, two communication rounds are required for reading from *regular* registers [GV06]. Dobre *et al.* [DMSS09] circumvented this lower bound by allowing the use of secret tokens. They give an implementation of a *regular* register where READ operations finish after a single communication round, if readers may write. However, scalability in the number of readers requires the READ operations to perform two communication rounds.

In the context of robust *atomic* register implementations, recent work of Dobre *et al.* [DGM⁺11] showed the impossibility of reading in two communication rounds. If WRITE operations perform a constant number of rounds, i.e., independent from the number of Byzantine faulty servers t , even reading in three rounds is impossible. That is why the number of communication rounds required for READ operations in existing atomic register implementations is either unbounded [AAB07, GLV06, GV07] or dependent on the number of faulty servers t [MAD02].

Several existing protocols aim for an optimal number of communication rounds in the best-case, i.e., assuming some degree of synchrony, a fewer number of faulty servers, or no concurrency among READ and WRITE operations. Under this assumptions, single-round Byzantine fault-tolerant atomic READ and WRITE are possible in the best-case [GLV06]. Work of Guerraoui and Vukolic [GV07] introduced for synchronous and uncontended runs single-round Byzantine fault-tolerant atomic READ and WRITE operations that gracefully degrade to two or three rounds depending on the number of available, correct servers. In the context of crash-fault tolerance, Georgiou *et al.* [GNS09] presented a *semifast* atomic read implementation.

Besides the large number of works in the area of Byzantine fault-tolerant storage that are based on the *unauthenticated data model* [MAD02, BD04, GWGR04, AAB07, ACKM06, ACKM07, GV06, GLV06, GV07, HGR07, CGK07, DMS08], some works assume *self-verifying data*, also called *authenticated* [MR98, DGLV05, CT06, LR06]. However, from a perspective of designing *lightweight* implementations, the use of self-verifying data [RSA78] is considered a significant source of overhead [Rei94, MR97] and is therefore not desirable.

As in work of Dobre *et al.* [DMSS09], the system model underlying this chapter of the thesis at hand, assumes the existence of a pseudo random number generator whose output may not be predicted by the malicious servers and clients. Using standard transformations from *regular* to *atomic* registers to combine the single-round regular READ operation of Dobre *et al.* [DMSS09] with the two-round WRITE operation of Abraham *et al.* [ACKM06], would result in a robust Byzantine fault-tolerant atomic register implementation where WRITE operations finish after two and READ operations finish after three rounds. Note, that this is the optimal

solution that can be achieved in the unauthenticated data model. To support an unbounded number of readers, the READ implementation of Dobre *et al.* requires two communication rounds, i.e., four rounds for the corresponding *atomic* READ operation. In this context, the robust atomic register implementation proposed in this chapter as the LWR protocol (C5), where both READ and WRITE operations require only two communication rounds, constitutes a significant reduction of the number of communication rounds compared to existing works. The LWR reduces the fundamental gap in the number of communication rounds between the authenticated data model (1-round write, 2-round read) [MR98] and the non-authenticated data model (2-round write, 4-round read) [DGM⁺11]. Furthermore, it constitutes the first atomic register implementation not relying on self-verifying data that tolerates an unbounded number of malicious readers [LR06].

Key-value store Besides the importance of fault-tolerant storage for the scientific community, registers are also a vital building block for practical distributed storage and file systems [SH02, SFV⁺04]. A similar trend can be observed for storage services in the “cloud” [CKS09b, MG11], that provide a simple read/write interface comparable to that of registers. Such a simple interface is provided by a *key-value store* (KVS), that has been formally defined for the first time by Basescu *et al.* in recent works [BCE⁺11, BCE⁺12]. Their work shows the close relation of the two abstractions register and KVS, as they introduce a fault-tolerant multi-writer multi-reader atomic register implemented on top of a collection of KVSs.

The KVS interface is implemented in the products of several today’s cloud service providers: Amazon’s key-value store Dynamo [DHJ⁺07] or the storage service S3 [AWS]; the storage systems Niobe [MTJ⁺08], Pahoehoe [ALM⁺10], and Cassandra [LM10]; Microsoft’s cloud storage service Windows Azure [CWO⁺11].

5.3 System Model

In this chapter we consider a distributed system as defined in Section 2.1.1. Here, the set of servers \mathcal{S} contains $n = 3t + 1$ servers. There are m clients $C = \{C_1, \dots, C_m\}$ in set \mathcal{C} . We define two subsets R and W of set \mathcal{C} denoted as *readers* and *writers*, such that $R \cap W = \emptyset$. Let the cardinality of the set of writers W be $|W| = n_w$. The writers may fail by crashing but never deviate from their specified behavior. Up to t servers and an arbitrary number of readers may be Byzantine faulty [PSL80], exhibiting non-responsive-arbitrary faults [JCT98].

Clients communicate with the servers by sending messages over reliable channels, forming an asynchronous network. Servers are unable to communicate with each other. Servers do not send messages to clients besides in response to client messages [ACKM06]. We assume that the channels between the writers and the

servers are *authenticated*, i.e., whenever some server receives a message from any writer w , it can be sure that the message has actually been sent by w . For the implementation variant (S) of the commitment scheme based on secret sharing, the channels between writers and servers are additionally assumed to be *secret*, i.e., if writer w sends a message to a correct server, then no malicious server or reader is able to read the message content. In a practical setting the two assumptions may be implemented using symmetric encryption or message authentication codes (MAC).

This chapter introduces the two distributed protocols LWR and LWKVS that implement the shared functionality of an *atomic register* and a *key-value store (KVS)* [BCE⁺12], respectively, in a wait-free manner (see Chapter 2 for the formal definitions). The time-complexity of the protocols is measured in terms of communication round-trips [LS02, DGLC04, EGM⁺09, GNS09, DGM⁺11]. During a *round-trip* (or *round*) the following steps are taken: (1) Some client sends messages to a subset of the servers; (2) the servers upon reception of a message reply back to the client before receiving any other messages; (3) the round terminates as soon as the invoking client receives enough replies.

We assume that the writers have access to a pseudo random number generator providing function `GETRANDOM()` which does not take any arguments and outputs a value in \mathbb{N} . The `GETRANDOM()` function satisfies the following *secrecy* property for any writer $w \in W$: No malicious server or reader is able to generate the k th output of `GETRANDOM()` before writer w invokes `GETRANDOM()` for the k th time.

For the commitment scheme variant (H) based on cryptographic hashing, we additionally assume that there exists a collision-resistant hash function H . Collision-resistance denotes that it satisfies the following property: Given any input x for the hash function H the malicious servers and readers are unable to generate some input x' such that $H(x) = H(x')$. The second implementation variant for the commitment scheme (S) proposed in this chapter makes use of a variation of Shamir's secret sharing scheme [Sha79]: Let a_0, a_1, \dots, a_t be the $t + 1$ coefficients of a polynomial of degree t . Then the commitment scheme satisfies the following property: t malicious servers are unable to generate all coefficients of the polynomial from only t points on the polynomial (i.e., for $i = 1, \dots, t$, (x_i, y_i) is a point on the polynomial iff $y_i = a_0 + x_i a_1 + x_i^2 a_2 + \dots + x_i^t a_t$).

5.4 (C5): The LWR Protocol

5.4.1 Idea of the LWR Protocol


Before introducing the ideas of the new lightweight atomic register implementation, let us recall the principles underlying existing robust register implementations

[ACKM06], which will then lead the discussion to the ideas and solutions behind the LwR protocol.

The predicate “safe” To ensure *regular* semantics, each value v that is written is assigned a timestamp by the writer; the timestamp is incremented with every new WRITE operation. Let us consider a run of the protocol where some operation $\text{WRITE}(v)$, where the writer assigns timestamp ts to value v , precedes some operation $\text{READ}()$. Regularity requires the $\text{READ}()$ operation to return v or a value with higher timestamp. In order not to block, the writer might access in the worst case only $n - t = 2t + 1$ servers, i.e., $t + 1$ *correct* ones³. Even if the reader, during operation $\text{READ}()$, accesses all t Byzantine servers and the remaining t correct ones that have not been accessed by the writer, the quorums of the reader and the writer intersect in at least one correct server holding (ts, v) . Hence, it would be sufficient for the reader to return the value with the highest timestamp. However, the reader has to make sure not to return a value that has been fabricated by a Byzantine server and that was never written. The principle is therefore only to return a value if it has been reported by $t + 1$ servers, i.e., by at least one correct server. In the LwR (and also LwKVS) protocol, this idea is captured by the predicate *safe*.

Necessity of two rounds How does a distributed protocol guarantee that the reader always receives $t + 1$ consistent replies? In the afore-described run, as (ts, v) is stored on $t + 1$ correct servers, the reader could simply wait for enough replies. However, there could be a run which is indistinguishable for the reader where the WRITE operation is still in progress: The reader has received replies from all correct servers, thus, cannot wait for more replies, and (ts, v) has not yet been written to $t + 1$ servers by the $\text{WRITE}(v)$ operation. Hence, in order not to block, the reader has to invoke another round of the $\text{READ}()$ operation. However, this might not help in a run where the writer crashes during operation $\text{WRITE}(v)$, as the reader would not receive any new information in the second round of $\text{READ}()$ (see Figure 5.1, execution in the middle). Therefore, to tolerate the crash of the writer, WRITE operations are performed in two rounds. The complete reasoning is illustrated by Figure 5.1⁴ showing three executions with four servers out of which one is Byzantine faulty: In the execution on the left, $\text{READ}()$ has to return value v as $\text{WRITE}(v)$ precedes $\text{READ}()$. However, the malicious server S_2 does

³Note, that robust implementations require optimal resilience which means that only $n = 3t + 1$ replicas can be used to tolerate t Byzantine-faulty ones.

⁴A communication round of a READ or WRITE operation is depicted as blocks arranged in a single column. In the column corresponding to some round of an operation, a block is drawn in a given row, if the corresponding server has received the message from the client in that round and has sent a reply message. A malicious server is marked by “”.

not show value v to the reader. The execution in the middle is indistinguishable for the reader, but here, the reader cannot wait for another reply as server S_1 is malicious and might be unresponsive. In the execution on the right, $\text{READ}()$ must not return value v as it has not been written but was fabricated by the malicious server S_3 . However, the reader cannot distinguish any of the three executions which contradicts the existence of single-round WRITE operations.

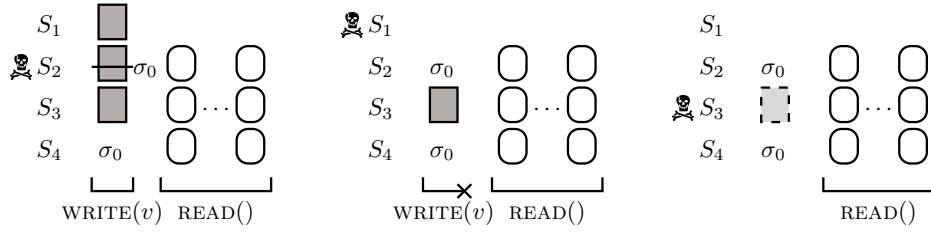



Figure 5.1: Impossibility of writing in one round. Symbol “” denotes a Byzantine faulty server and σ_0 the initial state of a server.

The two rounds of a $\text{WRITE}(v)$ operation are denoted as *pre-write* and *write* phase, the information sent in the pre-write (write) phase as pw (w). The reader collects in the first round of operation $\text{READ}()$ the w information from the servers as candidates to be returned. If it has read w from a correct server, then the corresponding pre-write phase is already complete. This implies, that in the second read round the reader can wait for the corresponding pw from $t + 1$ correct servers. Hence, the value corresponding to pw and w is **safe** and can be returned if there is no **safe** value with a higher timestamp.

From “regular” to “atomic” Obviously, such a READ implementation does not satisfy *atomicity*: Consider a run where $\text{READ}()$ returns a **safe** value v . In the most extreme case the corresponding candidate w and t replies containing pw in the second read round have been fabricated by Byzantine servers; i.e., value v is stored on only *one* correct server. Hence, another operation READ' succeeding READ might miss this correct server and not return v , violating *atomicity*. Therefore, the solution is for a robust *atomic* register implementation to let the reader, before returning value v , write value v back to the servers (using operation $\text{WRITE}(v)$). Hence, whenever $\text{READ}()$ returns v , v has been completely written such that succeeding READ operations may return v or a value with a higher timestamp. Intuitively, this explains why in existing Byzantine fault-tolerant robust atomic register implementations READ operations require four communication rounds (two for the actual READ and two for the WRITE -back).

Commitment scheme The idea behind the LWR protocol that implements an atomic register is to employ a *commitment scheme* [BCC88] to reduce the number of READ rounds from four to two. The simplest form of a cryptographic commitment scheme is a protocol between a sender and a receiver consisting of two phases. In the first, the *commit* phase, the sender creates a *commitment* from a secret value, called *token*, and sends it to the receiver. By the *hiding* property of the commitment, the chosen token cannot be known by the receiver. In the second, the *reveal* phase, the sender sends an *opening* to the receiver revealing the token. The *binding* property of the commitment ensures that the receiver can validate only the chosen token from the *opening* and the *commitment*.

During the *pre-write* phase of a $\text{WRITE}(v)$ operation, the writer chooses a random *token* and generates a *commitment* from it. The *commitment* is written together with the usual *pw* information to the servers. At the end of the pre-write, only the writer knows the *token*. In the write-phase, the writer sends *w* together with the chosen *token* as an opening to the servers.

New insights for the reader The use of a commitment scheme fundamentally changes the situation during READ operations. In the first round, the reader collects *w* information including *tokens* as candidates. If such a candidate *c* has actually been written, then in the second read round, $t+1$ correct servers will reply with *pw* including a *commitment* matching the *token* of candidate *c* and *c* is **safe**. This implies that for each **safe** candidate, the reader knows that the corresponding pre-write phase has been completed and that the corresponding *pw* is stored on $t+1$ correct servers. The reason is as follows: If the corresponding pre-write phase was not complete, by the *hiding* property of the commitment, no server would know the *token* corresponding to the *commitment* of the pre-write. However, if *commitment* and *token* match, then the *token* must have been actually written during the write phase (*binding* property), implying that the pre-write phase is already complete.

This property “if some candidate is **safe** then the corresponding *pw* is stored on $t+1$ correct servers” helps to simplify the write-back that ensures *atomicity*. The reader can be sure that these $t+1$ correct servers will reply to future READ' operations. Hence, the READ operation requires only one round to write back the candidate it is going to return. Therefore, the number of communication rounds for the READ has been reduced to three rounds.

Removing “incomplete” candidates Before explaining how to further reduce the number of READ rounds in the LWR protocol, we first consider the question how the reader deals with candidates that have been fabricated by Byzantine servers. Let *c* be such a candidate (including some *token*) that is reported in the first

read round only by Byzantine servers. Note, that the probability that the writer will choose the same *token* and construct a matching *commitment* is negligibly small. This implies that no correct server holds *pw* information containing a *commitment* that matches the *token* of the fabricated candidate *c*. Thus, if the reader receives in the second read round *pw* information from $2t + 1$ servers that does *not* correspond to *c* (which is covered by predicate *incomplete*), then the reader may ignore candidate *c* (i.e., remove it from the list of candidates). However, for candidates that actually have been written, this condition is never true: Then, $t + 1$ correct servers will reply with matching *pw* information in the second read round, i.e., at most $2t$ reply with non-matching *pw*.

Two read rounds The principle that the reader removes *incomplete* candidates during the second read round illustrates that each READ operation is able to distinguish written from fabricated candidates. Hence, to further reduce the number of read rounds, the reader does not need to wait until it determines the candidate to be returned before it writes-back: The idea is that after collecting candidates in the first read round (possibly including fabricated ones), the reader writes-back its whole *list* of candidates in the second read round. If it returns a value associated with some candidate *c*, then *c* is contained in the list of candidates and it is therefore written-back. An operation READ' succeeding READ will find candidate *c* on at least one server in the first round, i.e., it becomes also candidate for READ'. As *c* was *safe* during READ, the corresponding *pw* is stored on $t + 1$ correct servers, that will eventually reply in the second round of READ' such that *c* becomes *safe* also during READ'. By this, *atomicity* is guaranteed although the READ operations require only two communication rounds.

The handling of fabricated candidates also explains why the proposed atomic register implementations can tolerate an unbounded number of malicious readers. The only harm a malicious reader can do is to write-back a forged list of candidates. If it introduces new, fabricated candidates, then they are sorted out by other READ operations. As *atomicity* does not require anything from operation READ' succeeding operation READ of a malicious reader, it does not matter if a malicious reader writes-back an *incomplete* list of candidates.

Implementation of the Commitment Scheme The proposed LWR protocol is given in two variants, each implementing the commitment scheme differently. The first variant (H) is based on a collision-free hash function *H*. In this variant, the *token* is a random number chosen by the writer. The *commitment* is determined as $H(token)$. By the properties of the hash function *H*, *token* cannot be generated from *commitment*, and there exists no other *token'* that corresponds to *commitment*.

Variant (S) is based on ideas of Shamir's secret sharing scheme [Sha79]. Here, the *token* is a polynomial of degree t . Each server is assigned a different *commitment*, where each *commitment* is a point on the polynomial. As $t + 1$ points are required to determine a polynomial of degree t , the t points held by the Byzantine servers are not sufficient to determine the *token*. On the other hand, more than t *commitments* uniquely identify the secret polynomial, i.e., the *token*. Hence, for variant (S) it is important to note that communication between writer and servers has to be done via *secret channels*, as otherwise, Byzantine servers would be able to eavesdrop a number of *commitments* sufficient to determine the secret polynomial before it is revealed. However, in contrast to variant (H), variant (S) is *information theoretically secure* [LCAA07, AACL07].

5.4.2 Description of the LwR Protocol

The LwR protocol is described in two different variants. Variant (H) is based on a cryptographic hash function, while variant (S) makes use of a variation of Shamir's secret sharing scheme [Sha79]. The protocol is given as Algorithms 5.1, 5.2, and 5.3 for the WRITE operation, Algorithm 5.4 gives the implementation of the READ operation, and Algorithms 5.5, 5.6, and 5.7 specify the algorithm of the servers.

Algorithm 5.1: LwR Protocol, Algorithm of the Writer

```

Variables and Initialization:
     $ts$  integer, initially 0
     $pw$  structure  $\langle commitment, value \rangle$ 
     $w$  structure  $\langle ts, token \rangle$ 

WRITE( $v$ )
    /* Pre-write Phase */
5.1.1    CONSTRUCTTOKEN()
5.1.2     $ts \leftarrow ts + 1$ 
5.1.3    forall servers  $i = 1, \dots, n$  do
5.1.4         $pw.value \leftarrow v$ 
5.1.5         $pw.commitment \leftarrow \text{GETCOMMITMENT}()$ 
5.1.6        send  $pw\langle ts, pw \rangle$  to server  $i$ 
5.1.7    wait for reception of  $pwr\_ack\langle ts \rangle$  from  $n - t$  servers
    /* Write Phase */
5.1.8     $w.ts \leftarrow ts$ 
5.1.9     $w.token \leftarrow \text{GETOPENING}()$ 
5.1.10   send  $wr\langle ts, w \rangle$  to all servers
5.1.11   wait for reception of  $wr\_ack\langle ts \rangle$  from  $n - t$  servers
5.1.12   return OK

```

Write operation The write operation executes two rounds of communication, the *pre-write* phase and the *write* phase. After call $\text{WRITE}(v)$, initiating the pre-write phase, the writer generates a token by calling the $\text{CONSTRUCTTOKEN}()$ function. For variant (H), this function generates one random number, while for variant (S) $t + 1$ random numbers are generated. In both cases, the numbers generated by $\text{CONSTRUCTTOKEN}()$ are called the *token*. Next, the writer increases its timestamp, which is sent together with variable pw in a pre-write message to all servers. Variable pw is a structure consisting of value v and a *commitment*, generated by function $\text{GETCOMMITMENT}()$. In case (H), the *commitment* is the hash value of the constructed token. In case (S), function $\text{GETCOMMITMENT}()$ takes the $t + 1$ random numbers, generated before, as the $t + 1$ coefficients of a polynomial of degree t , and determines for each server a different, random point on the polynomial as the server's *commitment*. To complete the pre-write phase, the writer waits for acknowledgements from $n - t$ servers.

At the beginning of the write phase, the writer assigns its timestamp ts to entry $w.ts$ of variable w . Function GETOPENING is called to assign the *token*, generated in the pre-write phase to $w.token$, completing the update of variable w . Finally, the writer sends timestamp ts and variable w in a write-message to all servers and waits for $n - t$ acknowledgements before $\text{WRITE}(v)$ operation finishes.

Algorithm 5.2: Writer Procedures with Secret Sharing (S)

```

Variables and Initialization
   $x, a_1, a_2, \dots, a_t$  integers, initially 0
   $\text{CONSTRUCTTOKEN}()$ 
5.2.1   forall  $j = 0, \dots, t$  do
5.2.2      $a_j \leftarrow \text{GetRandom}()$ 
         $\text{GETCOMMITMENT}()$ 
5.2.3    $x \leftarrow \text{GetRandom}()$ 
5.2.4   return  $(x, a_0 + xa_1 + x^2a_2 + \dots + x^ta_t)$ 
         $\text{GETOPENING}()$ 
5.2.5   return  $(a_0, a_1, \dots, a_t)$ 

```

Read operation Each $\text{READ}()$ operation returns after two communication rounds. At beginning of the first round, the reader increments its reader timestamp tsr and sends it in a *read1* message to all servers. It waits for $n - t$ replies. Each such reply message from server i contains a set of candidates C_i which is added to set C before the reader finalizes the first round.

In the second round, the reader sends timestamp tsr and set C in a *read2* message to all servers. It waits for at least $n - t$ *read2_ack* messages as replies and until there is a candidate in set C satisfying both predicates *safe* and *highCand*.

Algorithm 5.3: Writer Procedures with Cryptographic Hash Function (H)

```

Variables and Initialization
    token integer, initially 0
CONSTRUCTTOKEN()
5.3.1    token  $\leftarrow$  GetRandom()
        GETCOMMITMENT()
5.3.2    return  $H(token)$ 
        GETOPENING()
5.3.3    return token

```

Whenever, a *read2_ack* messages is received from server i , the contained set of timestamp-value pairs TV is stored into $TV[i]$, while the server identifier i is added to set Q to keep track of the servers that have already replied. Upon each reception of a *read2_ack* message the reader checks for candidates $c \in C$ that satisfy predicate $\text{incomplete}(c)$ and removes such c from set C .

A candidate c satisfies predicate $\text{highCand}(c)$ if there is no other candidate in set C that has a higher timestamp than c . A candidate c is defined to be **safe** if $t + 1$ servers reply in the second round with the same timestamp-value pair $\langle ts, v \rangle$ such that ts matches $c.ts$, the timestamp of candidate c . Value v is denoted as the value *corresponding* to candidate c . A candidate c is called **incomplete** if $n - t$ servers return only timestamp-value pairs with timestamps different from $c.ts$.

Finally, the value corresponding to candidate c , for which $\text{safe}(c)$ and $\text{highCand}(c)$ holds, is returned.

Algorithm of the servers The basic operation of server i is to wait for the reception of a message from some client, to perform local computations and to send the corresponding reply back to the client. In total, there are messages of four different types: *write* and *pre-write* messages from the writer, and *read1* and *read2* messages from the readers. As the main local variables, server i maintains a variable w to store the content of *write* messages and a vector *history* with an entry for each possible timestamp for the handling of *pre-write* messages.

As soon as server i receives a *pre-write* message $pw\langle ts', pw \rangle$ from the writer, it assigns $\text{history}[ts'].pw$ field with pw and replies back to the writer by sending the acknowledgement $pwr_ack\langle ts' \rangle$.

If server i receives a *write* message $wr\langle ts', w' \rangle$ from the writer, it first checks whether ts' is greater than its own timestamp $w.ts$. If this is true, w' is assigned to variable w . Finally, server i replies with message $wr_ack\langle ts' \rangle$ to the writer.

Whenever server i receives a *read1* message $rd1\langle tsr \rangle$ from some reader k , it first sets set C to $\{w\}$. Next, it adds the content of all non-empty $\text{history}[ts'].wb$ fields to set C where $ts' > w.ts$. Then, server i sends C in a *read1_ack* message

Algorithm 5.4: LWR Protocol, Algorithm of the Readers

Variables and Initialization:

tsr integer, initially 0
 Q set of integers, initially \emptyset
 $TV[1..n]$ vector of sets of structure $\langle ts, value \rangle$, initially \emptyset
 C set of structure $\langle ts, token \rangle$, initially \emptyset
 c_{ret}, c structure $\langle ts, token \rangle$
 v, v_{ret} values

Predicates:

$highCand(c) \triangleq c \in C : (\forall c' \in C : c.ts \geq c'.ts)$
 $safe(c) \triangleq |\{i \in Q : \langle c.ts, v \rangle \in TV[i]\}| \geq t + 1$
 $incomplete(c) \triangleq |\{i \in Q : \langle c.ts, v \rangle \notin TV[i]\}| \geq n - t$

READ()

```

5.4.1   $C \leftarrow Q \leftarrow \emptyset$ 
5.4.2   $TV[i] \leftarrow \emptyset, 1 \leq i \leq n$ 
        /* Round 1
5.4.3   $tsr \leftarrow tsr + 1$ 
5.4.4  send  $rd1\langle tsr \rangle$  to all servers
5.4.5  repeat
5.4.6    if received  $rd1\_ack\langle tsr, C_i \rangle$  from server  $i$  then
5.4.7       $C \leftarrow C \cup C_i$ 
5.4.8  until received  $rd1\_ack\langle tsr, * \rangle$  from  $n - t$  servers
        /* Round 2
5.4.9  send  $rd2\langle tsr, C \rangle$  to all servers
5.4.10 repeat
5.4.11   if received  $rd2\_ack\langle tsr, TV \rangle$  from server  $i$  then
5.4.12      $Q \leftarrow Q \cup \{i\}; TV[i] \leftarrow TV$ 
5.4.13      $C \leftarrow C \setminus \{c \in C : incomplete(c)\}$ 
5.4.14   until (received  $rd2\_ack\langle tsr, * \rangle$  from  $n - t$  servers)  $\wedge$ 
           ( $\exists c \in C : safe(c) \wedge highCand(c)$ )
5.4.15   return VALUEOF( $c$ )

VALUEOF( $c$ )
5.4.16   $c_{ret} \leftarrow c : c \in C \wedge safe(c) \wedge highCand(c)$ 
5.4.17   $v_{ret} \leftarrow v : |\{i \in Q : \langle c_{ret}.ts, v \rangle \in TV[i]\}| \geq t + 1$ 
5.4.18  return  $v_{ret}$ 

```

$rd1_ack\langle tsr, C \rangle$ back to reader k .

Upon reception of a $read2$ message $rd2\langle tsr, CC \rangle$ from reader k , server i resets the set of timestamp-value pairs TV . Next, server i moves through all candidates c from received set CC and performs the following actions: It first adds c to set $history[ts].wb$ such that timestamp $ts = c.ts$. Then, if procedure $VERIFY(c)$ returns boolean value $true$, server i determines a timestamp-value pair from timestamp $c.ts$ and value $history[c.ts].pw.value$ and adds this timestamp-value pair to set TV . In case (H), procedure $VERIFY(c)$ returns boolean value $true$, if the hash value of token $H(c.token)$ equals the commitment $history[c.ts].pw.commitment$. In case (S), during procedure $VERIFY(c)$, the $t + 1$ coefficients (a_0, \dots, a_t) of the polynomial of degree t are extracted from token $c.token$. Further, the coordinates of point (x, y) are assigned the content of $history[c.ts].pw.commitment$. $VERIFY(c)$ returns $true$, if point (x, y) lies on the polynomial defined by coefficients (a_0, \dots, a_t) .

Finally, set TV is sent in a $rd2_ack\langle tsr, TV \rangle$ message back to reader k .

5.4.3 Proof of Correctness of the LWR Protocol

We show in this section that the LWR protocol given as Algorithms 5.1, 5.2, 5.3, 5.4, 5.5, 5.6, and 5.7 implements a single-writer multiple-reader atomic register.

The next corollary shows that regularity between $WRITE$ and $READ$ operations is satisfied. This means that a $READ$ operation does not return a value which is older than the one written by the last preceding $WRITE$ operation.

Corollary 5.1. Let $WRITE(v)$ be a write operation that writes value v with timestamp ts_v . Let $READ()$ be a read operation of reader k such that $WRITE(v)$ precedes $READ()$. Then, operation $READ()$ does not return a value with timestamp $< ts_v$.

Proof. In the first round of operation $READ()$, reader k receives candidates from variable w of $2t + 1$ servers (lines 5.5.9 and 5.5.12). Reader k collect these candidates in set C (line 5.4.7). After the first round of $READ()$, let w' be the candidate with the highest timestamp in set C at reader k , which has been sent by a correct server. We first show that $w'.ts \geq ts_v$:

As the operation $WRITE(v)$ is complete, it has sent w_v where $w_v.ts = ts$ to $2t + 1$ servers during the write-phase (line 5.1.10). Hence, at least $t + 1$ correct servers have received this $wr\langle ts_v, w_v \rangle$ message from reader k . For these servers holds upon the reception of the $write$ message: Either $ts_v \leq w.ts$, in which case some other candidate w' with timestamp ts' has already been received such that $ts' = w.ts \geq w_v.ts$. Or, $w_v.ts > w.ts$, in which case candidate w_v is stored to the local variable w (line 5.5.6). Thus, at least one correct server adds w_v or a candidate with higher timestamp to its local set C (line 5.5.9) and sends it as a reply to the $read1$ message in a $read1_ack$ message to reader k (line 5.5.12). This

Algorithm 5.5: LWR Protocol, Algorithm of Server i

Variables and Initialization:
 w structure $\langle ts, token \rangle$, initially $\langle 0, token_0 \rangle$
 $history[.]$ vector of structure $\langle pw, wb \rangle$
 pw structure $\langle commitment, value \rangle$; wb set of structure $\langle ts, token \rangle$
 $history[j].wb$ initially \emptyset , $j \geq 0$
 $history[0].pw$ initially $\langle commitment_0, \perp \rangle$
 C set of structure $\langle ts, token \rangle$, initially \emptyset
 TV set of structure $\langle ts, value \rangle$, initially \emptyset

5.5.1 **upon** reception of $pw\langle ts', pw \rangle$ from writer
5.5.2 $history[ts'].pw \leftarrow pw$
5.5.3 send $pwr_ack\langle ts' \rangle$ to writer
5.5.4 **upon** reception of $wr\langle ts', w' \rangle$ from writer
5.5.5 **if** $ts' > w.ts$ **then**
5.5.6 $w \leftarrow w'$
5.5.7 send $wr_ack\langle ts' \rangle$ to writer
5.5.8 **upon** reception of $rd1\langle tsr \rangle$ from reader k
5.5.9 $C \leftarrow \{w\}$
5.5.10 **forall** $ts' > w.ts$ s.t. $history[ts'].wb \neq \emptyset$ **do**
5.5.11 $C \leftarrow C \cup history[ts'].wb$
5.5.12 send $rd1_ack\langle tsr, C \rangle$ to reader k
5.5.13 **upon** reception of $rd2\langle tsr, CC \rangle$ from reader k
5.5.14 $TV \leftarrow \emptyset$
5.5.15 **forall** $c \in CC$ **do**
5.5.16 $history[c.ts].wb \leftarrow history[c.ts].wb \cup \{c\}$
5.5.17 **if** $VERIFY(c)$ **then**
5.5.18 $TV \leftarrow TV \cup \{\langle c.ts, history[c.ts].pw.value \rangle\}$
5.5.19 send $rd2_ack\langle tsr, TV \rangle$ to reader k

Algorithm 5.6: VERIFY Procedure with Hash Function (H)

$VERIFY(c)$
5.6.1 **return** $(H(c.token) = history[c.ts].pw.commitment)$

Algorithm 5.7: VERIFY Procedure with Secret Sharing (S)

$VERIFY(c)$
5.7.1 $(x, y) \leftarrow history[c.ts].pw.commitment$
5.7.2 $(a_0, \dots, a_t) \leftarrow c.token$
5.7.3 **return** $(y = a_0 + xa_1 + x^2a_2 + \dots + x^ta_t)$

means that reader k adds w_v (or a candidate with higher timestamp) to set C (line 5.4.7). Thus, the claim $w'.ts \geq ts_v$ is true.

Next, we show that during operation $\text{READ}()$ reader k never removes candidate w' from set C . Let $\text{WRITE}(v')$ be the operation writing candidate w' to a correct server, i.e., let v' be the value written by operation $\text{WRITE}(v')$ with timestamp $w'.ts \geq ts_v$. We continue the proof separately for the two variants (H) and (S):

Hash Function, Case (H). In the pre-write-phase of operation $\text{WRITE}(v')$, the writer has sent $\langle w'.ts, (H(w'.token), v') \rangle$ to $2t + 1$ servers (line 5.1.6), where $w'.token$ is the output of $\text{GETRANDOM}()$ when called by the writer during operation $\text{WRITE}(v')$ (line 5.3.1). As w' is stored in the local variable w of some correct server, the pre-write phase of $\text{WRITE}(v')$ is complete. Hence, at least $t + 1$ correct servers hold $\langle H(w'.token), v' \rangle$ in their $\text{history}[w'.ts].pw$ field (line 5.5.2). Thus, call of $\text{VERIFY}(w')$ (line 5.5.17) at any of these $t + 1$ correct servers would return *true* (line 5.6.1), and timestamp-value pair $\langle w'.ts, v' \rangle$ is contained in these servers' *read2_ack* messages (lines 5.5.18 and 5.5.19). This implies that there are at most $2t$ servers that reply with timestamp-value pairs different from $\langle w'.ts, v' \rangle$ to *read2* messages of reader k .

Secret Sharing, Case (S). In the pre-write-phase of operation $\text{WRITE}(v')$, the writer has sent $\langle w'.ts, (x_i, a'_0 + x_i a'_1 + x_i^2 a'_2 + \dots + x_i^t a'_t), v' \rangle$ to server i of a set of $2t + 1$ servers (line 5.1.6), where a'_0, a'_1, \dots, a'_t and x_i is the output of $\text{GETRANDOM}()$ when called during $\text{WRITE}(v')$ (lines 5.2.2 and 5.2.3). Note that $w'.token = (a'_0, a'_1, \dots, a'_t)$. As w' is stored in the local variable w of some correct server, the pre-write phase of $\text{WRITE}(v')$ is complete. Hence, at least $t + 1$ correct servers i hold $\langle w'.ts, (x_i, a'_0 + x_i a'_1 + x_i^2 a'_2 + \dots + x_i^t a'_t), v' \rangle$ in their $\text{history}[w'.ts].pw$ field (line 5.5.2). Thus, call of $\text{VERIFY}(w')$ (line 5.5.17) at any of these $t + 1$ correct servers would return *true* (line 5.7.3), and timestamp-value pair $\langle w'.ts, v' \rangle$ is contained in these servers' *read2_ack* messages (lines 5.5.18 and 5.5.19). This implies that there are at most $2t$ servers that reply with timestamp-value pairs different from $\langle w'.ts, v' \rangle$ to *read2* messages of reader k .

This implies that in both algorithm variants w' is never removed from the set of candidates C by reader k (line 5.4.13). Hence, predicate *highCand* makes sure that $\text{READ}()$ does not return a value with timestamp $< ts_v$. \square

An atomic register satisfies stronger consistency properties than a regular register. Thus, the next corollary proves that the LwR protocol additionally guarantees that a READ operation does not return a value which is older than the one returned by the last preceding READ , as required by the definition an atomic register.

Corollary 5.2. Let $\text{READ}'()$ be a read operation of *correct* reader k' that has returned value v (with timestamp ts_v). Let $\text{READ}()$ be a read operation of reader k such that $\text{READ}'()$ precedes $\text{READ}()$. Then, operation $\text{READ}()$ does not return a value with timestamp $< ts_v$.

Proof. As $\text{READ}'()$ has returned v there has been a candidate w_v during the second round of $\text{READ}'()$ that satisfied $\text{safe}(w_v)$ and $\text{highCand}(w_v)$ and where $w_v.ts = ts_v$. We first show that w_v has been written in the write-phase of some write operation $\text{WRITE}(v)$ and that the corresponding pre-write phase is complete. We proceed the proof separately for the two variants (H) and (S):

Hash Function, Case (H). By the definition of $\text{safe}(w_v)$ (in Algorithm 5.4), there exist $t + 1$ servers that have replied with read2_ack messages (line 5.5.19) containing $\langle ts_v, v \rangle$ to reader k' during operation $\text{READ}'()$. Hence, there is at least one correct server where call $\text{VERIFY}(w_v)$ returned *true* and timestamp-value pair $\langle ts_v, v \rangle$ was contained in this server's read2_ack message. Hence, one correct server holds pw in its $\text{history}[w_v.ts].pw$ such that $H(w_v.token) = pw.commitment$. By the hiding property of hash function H , any server can know $w_v.token$ only after it has been sent (line 5.1.10) during the write-phase of some write operation $\text{WRITE}(v)$ which further implies that the corresponding pre-write-phase is complete.

Secret Sharing, Case (S). By the definition of $\text{safe}(w_v)$ (in Algorithm 5.4), there exist $t + 1$ servers that have replied with read2_ack messages (line 5.5.19) containing $\langle ts_v, v \rangle$ to reader k' during operation $\text{READ}'()$. Hence, there is at least one correct server where call $\text{VERIFY}(w_v)$ returned *true* and timestamp-value pair $\langle ts_v, v \rangle$ was contained in this server's read2_ack message. Hence, one correct server i holds pw in its $\text{history}[w_v.ts].pw$ such that $(x_i, y_i) = pw.commitment$, $(a_0, \dots, a_t) = w_v.token$, and $a_0 + x_i a_1 + x_i^2 a_2 + \dots + x_i^t a_t = y_i$. By the hiding property of the secret sharing scheme, any server can know $w_v.token$ only after it has been sent (line 5.1.10) during the write-phase of some write operation $\text{WRITE}(v)$ which further implies that the corresponding pre-write-phase is complete.

Note, that in both cases, at the point in time when reader k' returns value v at the end of operation $\text{READ}'()$, the pre-write phase of $\text{WRITE}(v)$ is complete. Next, we show that during first round of $\text{READ}()$ reader k adds a candidate with timestamp $\geq ts_v$ to the set of candidates C which is never removed:

As v has been returned by $\text{READ}'()$, it must have been a candidate for reader k' , i.e., $w_v \in C$ during the first round of $\text{READ}'()$ at reader k' . As operation $\text{READ}'()$ is complete, set C containing w_v has been sent to $2t + 1$ servers in read2 messages. At least $t + 1$ correct servers receive w_v in such a read2 message. Each such server adds w_v its $\text{history}[w_v.ts].wb$ (line 5.5.16).

As $\text{READ}'()$ precedes $\text{READ}()$, reader k will receive set of candidates C_i containing w_v or a candidate with higher timestamp w' from at least one correct server i during first read-round. This is because server i either adds w_v from its $\text{history}[w_v.ts].wb$ to set C_i (line 5.5.11) or a candidate w' with higher timestamp than $w_v.ts$ (line 5.5.9). As the corresponding pre-write phases of w_v and w' (if it exists) are complete, by the same arguments as in the proof of Corollary 5.1, w_v or w' (if it exists) is never removed from set of candidates by reader k dur-

ing $\text{READ}()$. Thus by predicate highCand , $\text{READ}()$ does not return a value with timestamp $< ts_v$. \square

The next two corollaries show that the LWR protocol never returns a value that has been fabricated by a malicious server.

Corollary 5.3. No $\text{READ}()$ operation of a *correct* reader k returns a value with timestamp < 0 .

Proof. Initially, all correct servers store $\langle 0, token_0 \rangle$, called candidate c_0 in their variable w and $\langle commitment_0, \perp \rangle$ in $history[0].pw$. As the writer is correct and increases its timestamp at the beginning of a WRITE operation (line 5.1.2), w is only updated with candidates with higher timestamps. Hence, for any $\text{READ}()$ holds, that any correct server only proposes candidates with timestamp ≥ 0 . By the same arguments as in the proof of Corollary 5.1, such a candidate of a correct server is never removed from set of candidates by reader k during $\text{READ}()$. Thus by predicate highCand , $\text{READ}()$ does not return a value with timestamp < 0 . \square

Corollary 5.4. Let $\text{READ}()$ be a read operation of *correct* reader k that returns value $v \neq \perp$ (with timestamp ts_v). Then, there exists an operation $\text{WRITE}(v)$ such that $\text{READ}()$ does not precede $\text{WRITE}(v)$. If $\text{READ}()$ returns \perp (with timestamp 0), then there exists no $\text{WRITE}(v)$ operation that precedes $\text{READ}()$.

Proof. If $\text{READ}()$ returns value v , there exists a candidate $c \in C$ for reader k such that $\text{safe}(c)$ holds (line 5.4.14). This implies that there are $t + 1$ servers at which $\text{VERIFY}(c)$ returns *true* and that send timestamp-value pair $\langle ts_v, v \rangle$, $c.ts = ts_v$, in *read2_ack* messages to reader k . As there are at most t malicious servers, among these servers at least one server is correct. By line 5.5.18, the correct server has determined v as $history[c.ts].pw.value$. As $history[c.ts].pw.value \neq \perp$, the correct server has received a *pre-write* message of some $\text{WRITE}(v)$ operation containing pw such that $pw.value = v$. This *pre-write* message was received by the correct server *before* it received the *read2* message during $\text{READ}()$. Thus, $\text{READ}()$ does not precede $\text{WRITE}(v)$.

If $\text{READ}()$ returns value \perp , there exists a candidate $c_0 \in C$, $c_0.ts = 0$ for reader k such that $\text{safe}(c_0)$ and $\text{highCand}(c_0)$ holds (line 5.4.14) and $\text{VALUEOF}(c_0)$ returns \perp . This implies that for all candidates $c \in C$ such that $c.ts > 0 = c_0.ts$ holds $\text{incomplete}(c)$ and they have been removed from C (line 5.4.13). Let $\text{WRITE}(v)$ be the operation that has written such a candidate c during write-phase.

Predicate $\text{incomplete}(c)$ implies that $2t + 1$ servers have replied with *read2_ack* messages not containing any timestamp-value pairs $\langle c.ts, v \rangle$. Hence, at $t + 1$ correct servers, $history[c.ts].pw$ does not contain $\langle commitment_v, v \rangle$. This further implies that $t + 1$ correct servers did not receive a *pwr* $\langle ts_v, \langle commitment_v, v \rangle \rangle$ message

before they reply with *read2_ack* messages. Thus, the pre-write phase of $\text{WRITE}(v)$ is not complete when second round of $\text{READ}()$ is initiated, and therefore, $\text{WRITE}(v)$ does not precede $\text{READ}()$. \square

The next lemma completes the safety part of the correctness proof of the LWR protocol. Lemma 5.6 continues the proof by showing the liveness properties of the LWR protocol.

Lemma 5.5. The READ and WRITE operations implemented by the LWR protocol in Algorithm 5.1, 5.2, 5.3, 5.4, 5.5, 5.6, and 5.7 satisfies the safety properties of an atomic register.

Proof. Correctness follows directly from Corollaries 5.1, 5.2, 5.3, and 5.4. \square

Lemma 5.6. The READ and WRITE operation implemented by the LWR protocol in Algorithm 5.1, 5.2, 5.3, 5.4, 5.5, 5.6, and 5.7 are *wait-free*.

Proof. By assumption, there are at most t malicious servers. As the writer never waits for more than $n - t$ replies during pre-write and write phase of a WRITE operation and as the reader never waits for more than $n - t$ replies during the first round of a READ operation, it remains to show that no $\text{READ}()$ operation blocks during the second read round.

We assume for contradiction that some operation $\text{READ}()$ of correct reader k blocks during the second round, i.e., we consider the point in time where all correct servers have replied with *read2_ack* messages (line 5.4.14). We first show, that the set of candidates C at reader k is not empty. Therefore, we assume that w is the last candidate that has been written in line 5.5.6 (during write phase of some $\text{WRITE}(v)$ operation) to some correct server i (or the initial candidate c_0 if no candidates have been written yet) before server i replies to *read1* message during $\text{READ}()$ (line 5.5.12). Hence, w is contained in C_i received from server i during first round of $\text{READ}()$ and w is added to C by reader k (line 5.4.7). As the corresponding pre-write phase precedes $\text{READ}()$, w is never removed from set C (cf. proof of Corollary 5.1).

We proceed by showing that all candidates $c \in C$ are *safe*. For contradiction, we assume that there exists a candidate c contained in set C which is not *safe*. Candidate c has been sent by some servers in a *read1_ack* message to reader k during $\text{READ}()$ (line 5.5.12). At the end of the first round of $\text{READ}()$, there exists (Case 1) either some $\text{WRITE}(v)$ operation that has stored c during write-phase to some server, (Case 2) or there is no such operation as c has been fabricated by some malicious server.

Case 1 There exists a $\text{WRITE}(v)$ operation that has stored candidate c during write-phase onto some server. Hence, the corresponding pre-write phase has been

completed before the second round of $\text{READ}()$ is started. It directly follows that c is *safe*, as on $t + 1$ correct servers $\text{VERIFY}(c)$ returns *true* and these servers send *read2_ack* messages containing timestamp-value pair $\langle c.ts, v \rangle$, which is a contradiction.

Case 2 As candidate c has not been written at all, by the hiding property of the hash function and the secret sharing scheme, on $2t + 1$ correct servers $\text{VERIFY}(c)$ returns *false* and these servers send *read2_ack* messages not containing timestamp-value pair $\langle c.ts, * \rangle$. Thus, c is removed from set C (line 5.4.13).

Therefore, candidates in C are either *safe* or eventually removed from C . Thus, no $\text{READ}()$ operation blocks in the second round (line 5.4.14) and all implemented operations are *wait-free*. \square

The following theorem completes the correctness proof of the LwR protocol.

Theorem 5.7. Protocol LwR in Algorithms 5.1, 5.2, 5.3, 5.4, 5.5, 5.6, and 5.7 implements a single-writer multiple-reader atomic register in a wait-free manner.

Proof. The correctness follows directly from Lemmas 5.5 and 5.6 \square

5.5 (C6): The LwKVS Protocol

This section introduces the LwKVS protocol being a lightweight and robust Byzantine fault-tolerant implementation of a key-value store. The main principle of storing and retrieving data from the KVS is based on the ideas of the atomic register implementation given as the LwR protocol in section 5.4.

5.5.1 From Atomic Registers to Key-Value Stores

As already mentioned in the introduction in section 5.1, the interfaces of a KVS and an atomic register are very similar. Operations $\text{PUT}(key, v)$ and $\text{GET}(key)$ of a KVS are used to store and retrieve data from the KVS under a given key key . Hence, each key key could be modeled as a separate atomic register R_{key} , and $\text{PUT}(key, v)$ and $\text{GET}(key)$ could be emulated by atomic operations $\text{WRITE}(v)$ and $\text{READ}()$ accessing register R_{key} . Thus, the idea for this section is to reuse as much as possible of the implementation of the lightweight and robust atomic register from the LwR protocol for the implementation of the lightweight and robust KVS in the LwKVS protocol.

However, there are also important differences between the interface of an atomic register and a KVS. The atomic register implementation of protocol LwR in section 5.4 is based on the assumption, that such an atomic register is accessed by only *one* writer (single-writer multiple-reader setting). For the interface of a KVS it is required that each key can be accessed by *multiple* writers via operations

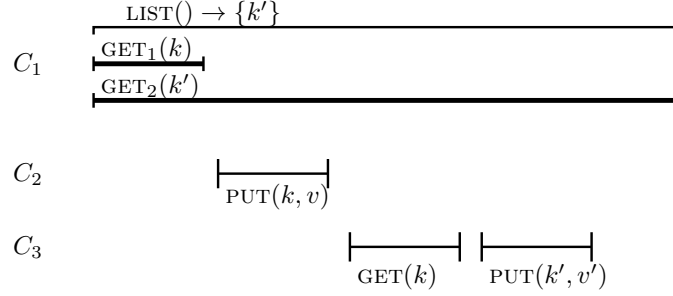
$\text{PUT}(key, *)$. Hence, the implementation of a KVS has to enable multiple writers: Intuitively, the timestamps are partitioned such that each *writer* is preassigned a disjunct subset of the timestamps. At invocation of a $\text{PUT}(key, v)$ operation, the writer performs an extra round of communication with the servers, requesting the highest timestamp ts_{key} used for key key so far. It then chooses a timestamp $> ts_{key}$ from its own set of timestamps as the new timestamp for operation $\text{PUT}(key, v)$. This extra round of communication is important to ensure *atomicity* among PUT and GET operations.

Secondly, a KVS provides a $\text{LIST}()$ operation returning a list of keys that are currently “in use”, which does not exist for atomic registers. Obviously, an implementation of such a $\text{LIST}()$ operation has to access all possible keys of a KVS to determine the set of keys that store a non- \perp value, i.e., the keys “in use”. Hence, the idea for the implementation of the $\text{LIST}()$ operation is to reuse the principles of the implementation of the $\text{GET}()$ operation. In order to keep the implementation lightweight it is important that the $\text{LIST}()$ operation, as operation GET, requires only two rounds of operation with the clients. To get along with only two communication rounds, the access to all possible keys during the LIST operation has to be done in parallel.

However, such an implementation of a LIST operation may violate the consistency property of *linearizability* as dictated by atomic operations. Intuitively, *linearizability* requires that in any execution of the protocol all PUT, DELETE and GET operations are either scheduled before or after each operation $\text{LIST}()$. The following counterexample illustrates that is is not possible with a two-round $\text{LIST}()$ operation that is implemented from a collection of $\text{GET}(key)$ operations accessing each possible key key . Let us now consider the following execution of the protocol illustrated in Figure 5.2: The implemented KVS provides only two possible keys k and k' and is accessed by three clients C_1 , C_2 and C_3 . Client C_1 executes a $\text{LIST}()$ operation that is implemented by two concurrent operations $\text{GET}_1(k)$ and $\text{GET}_2(k')$. Client C_2 stores value $v \neq \perp$ under key k , such that operation $\text{GET}(k)$ of client C_3 returns value v . Finally, client C_3 stores value $v' \neq \perp$ into the KVS under key k' . As during operation $\text{LIST}()$ of client C_1 , operation $\text{GET}_1(k)$ returns \perp and $\text{GET}_2(k')$ returns $v' \neq \perp$, the list of keys returned by client C_1 contains only key k' .

However, for the execution illustrated in Figure 5.2 there exists no *linearizable* order of the operations: As key k' is contained in the list of keys returned by operation $\text{LIST}()$, $\text{LIST}()$ must not be ordered before operation $\text{PUT}(k', v')$. On the other hand, as key k is not returned by operation $\text{LIST}()$, $\text{LIST}()$ must not be ordered after $\text{PUT}(k, v)$. As operation $\text{PUT}(k, v)$ precedes operation $\text{PUT}(k', v')$ it is easy to see that no such total order exists. This means that the implemented $\text{LIST}()$ operation, although implemented from atomic GET operations, violates *atomicity*.

The above argumentation shows that the main challenge for the implementation

Figure 5.2: Output of operation $\text{LIST}()$ violating atomicity

of the $\text{LIST}()$ operation is to find a good compromise between the provided degree of consistency and the complexity of the implementation itself. To achieve a fully consistent output of the $\text{LIST}()$ operations, in the execution of Figure 5.2, client C_1 would have to read from all keys again (in order not to miss key k written in the meantime by operation $\text{PUT}(k, v)$). However, even the next “wave” of GET operations during $\text{LIST}()$ might be interleaved with other PUT and GET operations such that the inconsistencies remain. Thus, to keep the implementation of the $\text{LIST}()$ operation lightweight and the number of communication rounds as low as possible, the idea for the implementation of the $\text{LIST}()$ operation of the LwKVS protocol is to surrender *atomicity* and to provide *regular* semantics instead.

From a practical point of view such *regular* $\text{LIST}()$ operations are attractive as they can be efficiently implemented and as the provided semantics are “consistent enough”: The purpose of the $\text{LIST}()$ operation is to provide to the client an overview of all keys under which non- \perp values are currently stored in the KVS. This means that if some client first uses operation $\text{LIST}()$ to determine set K of currently used keys, and then initiates operation $\text{GET}(k)$ for some key $k \in K$, the client expects to retrieve a non- \perp value from operation $\text{GET}(k)$. Informally, this behavior is ensured by *regular* LIST operations: If $\text{PUT}(k, v)$ precedes $\text{LIST}()$ and there is no operation deleting key k after $\text{PUT}(k, v)$, then key k is contained in the list of keys returned by operation $\text{LIST}()$. Hence, for a lightweight implementation of a fault-tolerant KVS, $\text{LIST}()$ operations that provide *regular* semantics, constitute a reasonable compromise between consistency and implementation complexity. As we will later see, the $\text{LIST}()$ operation requires only two rounds of communication between client and servers and several principles of the implementation of the *atomic* GET operation can be reused.

5.5.2 Description of the LwKVS Protocol

The protocol is described in two different variants. As in section 5.4, variant (H) is based on a cryptographic hash function, while variant (S) makes use of a variation of Shamir’s secret sharing scheme [Sha79]. The LwKVS protocol is given as Algorithm 5.8 for the PUT and DELETE operations of the *writers*. Procedures CONSTRUCTTOKEN, GETCOMMITMENT, and GETOPENING are given as Algorithms 5.2 and 5.3 in Section 5.4 and are used here without any changes. Algorithms 5.11, 5.12, 5.13, and 5.14 give the algorithms executed by the servers, while Algorithms 5.9 and 5.10 give the implementation of GET and LIST operations of the *readers*.

Operations PUT, GET, and DELETE are *atomic* operations, while operation LIST achieves *regular* semantics. Operation PUT requires three rounds of communication with the servers. The DELETE(key) operations are implemented as operations PUT(\perp, key). Operations GET and LIST finish after only two rounds of communication. All operations are wait-free. In the following only the differences with respect to the robust atomic register implementation of protocol LwR in Section 5.4 are described.

Local KVS The algorithm executed by the servers makes use of local key-value stores providing low-level operations Put and Get. The idea is that keys of the local KVS are a composition of the key of the implemented, high-level operation and protocol specific information, e.g., instead of storing a candidate $\langle ts, token \rangle$ to local variable w as done by the servers in the LwR protocol in section 5.4, here during high-level operation PUT(key, v), the corresponding candidate is stored in the local KVS by call Put($“w” \circ key, \langle ts, token \rangle$). Similar, pre-writes for key key and timestamp ts are stored under local-key $“pw” \circ key \circ ts$.

Note that if local Get(key) is called and the local KVS has not stored a value for key key , then Get(key) returns **fail**. However, to ease the presentation of the pseudo-code we make the following assumptions that operation call GET(key) as used in the pseudo-code never returns **fail**:

- If local KVS has no entry for key $“w” \circ key \circ ts$, then Get($“w” \circ key \circ ts$) returns value $\langle 0, key_0, token_0 \rangle$.
- If local KVS has no entry for key $“pw” \circ key \circ ts$, then Get($“pw” \circ key \circ ts$) returns value $\langle commitment_0, \perp \rangle$.
- If local KVS has no entry for key $“wb” \circ key \circ ts$, then Get($“wb” \circ key \circ ts$) returns value \emptyset .

Put and Delete operation. The PUT operation implemented by the LwKVS protocol is based on the atomic WRITE operation given in section 5.4 and also

performs a pre-write and a write phase. To handle multiple writers, it additionally executes a *read timestamp phase* prior to the *pre-write phase*. Therefore, at the beginning of a $\text{PUT}(key, v)$ operation, the writer sends key and its previous timestamp in a *timestamp_request* message to the servers. Each server responds with the highest timestamp of a candidate that has been stored for key key in a WRITE phase or a second round of a GET operation. The writer determines the highest timestamp ts' among the received timestamps and chooses a timestamp higher than ts' from its own set of timestamps: If there are n_w many writers in the system, then writer i , $i = 1, \dots, n_w$ is preassigned all timestamps ts such that $(ts \bmod n_w) = i - 1$ holds. Hence, the next higher timestamp than ts' satisfying this condition is computed as $ts' - (ts' \bmod n_w) + n_w + i - 1$ by writer i .

The pre-write and write phase are performed analogously to Algorithm 5.1 in section 5.4. Additionally all messages exchanged maintain key as an additional parameter. Operation $\text{DELETE}(key)$ is implemented by call $\text{PUT}(key, \perp)$.

Get and List operation The implemented $\text{GET}(key)$ operation executes in two rounds. Its implementation analogously follows the implementation of operation $\text{READ}()$ in Algorithm 5.4 from Section 5.4. The difference is that each message *read1*, *read1_ack*, *read2*, and *read2_ack* contains now key key as an additional parameter. A slight change has been made to the VERIFY procedure, that also takes key as an additional input parameter and returns a timestamp-value pair (or \emptyset) instead of true (false).

Just like GET operations, each LIST() operation finishes after two rounds. In the first round, the reader increments timestamp tsl which is sent in a *list1* message to all servers. The reader waits for $n - t$ replies *list1_ack* each containing a set of *list candidates*. A list candidate is a structure consisting of a timestamp, a key and a token (note that the normal candidates used during GET operations are a pair $\langle ts, token \rangle$). All list candidates are collected in set LC at the end of the first round. Different to the set C used during GET operation, set LC contains candidates for different keys.

The purpose of the second round is for the reader to eliminate list candidates that satisfy predicate *incomplete_ls* from set LC and wait for the remaining list candidates in LC to eventually become *safe_ls*. Hence, at the beginning of the second round, the reader sends set LC to all servers in *list2* messages. The servers reply with a set of triples $\langle ts, key, flag \rangle$. If some server i replies with triple $\langle ts, key, flag \rangle$ as response to list candidate lc , this means that procedure VERIFY at server i has returned timestamp-value pair (ts, val) for candidate $\langle lc.ts, lc.token \rangle$ and key key . The corresponding *flag* is true if $val = \perp$ and false otherwise.

The reader waits for at least $n - t$ *list2_ack* messages in the second round of LIST() and until all candidates in set LC satisfy predicate *safe_ls*. A list candidate lc is defined to be *safe_ls* if $t + 1$ servers reply in the second round with the same

Algorithm 5.8: LWKVS Protocol, PUT and DELETE Operations
(Algorithm of Client k)

Variables and Initialization:
 n_w integer, constant, number of writers
 ts, ts_{old} integer, initially 0
 pw structure $\langle commitment, value \rangle$
 w structure $\langle ts, token \rangle$

PUT(v, key)

```

/* Read-timestamp Phase */
5.8.1   $ts_{old} \leftarrow ts$ 
5.8.2  send  $ts\_req\langle key, ts_{old} \rangle$  to all servers
5.8.3  repeat
5.8.4    if received  $ts\_req\_ack\langle key, ts', ts_{old} \rangle$  from server  $i$ 
5.8.5    then
5.8.5    if  $ts' > ts$  then  $ts \leftarrow ts'$ 
until received  $ts\_req\_ack\langle key, *, ts_{old} \rangle$  from  $n - t$  servers
/* Pre-write Phase */
5.8.6   $ts \leftarrow ts - (ts \bmod n_w) + n_w + k - 1$  /* inc. timestamp */
5.8.7  CONSTRUCTTOKEN()
5.8.8  forall servers  $i = 1, \dots, n$  do
5.8.9     $pw.value \leftarrow v$ 
5.8.10    $pw.commitment \leftarrow GETCOMMITMENT()$ 
5.8.11   send  $pwr\langle key, ts, pw \rangle$  to server  $i$ 
5.8.12  wait for reception of  $pwr\_ack\langle key, ts \rangle$  from  $n - t$  servers
/* Write Phase */
5.8.13   $w.ts \leftarrow ts$ 
5.8.14   $w.token \leftarrow GETOPENING()$ 
5.8.15  send  $wr\langle key, w \rangle$  to all servers
5.8.16  wait for reception of  $wr\_ack\langle key, ts \rangle$  from  $n - t$  servers
5.8.17  return OK

DELETE( $key$ )
5.8.18  PUT( $\perp, key$ )
5.8.19  return OK

```

timestamp-key-flag triple $\langle ts, key, flag \rangle$ such that ts matches $lc.ts$. If $flag$ is true, lc satisfies `safe_ls_remove`.

Finally, the reader keeps for each possible key key only the list candidate with the highest timestamp in set LC (by using predicate `highCand_ls`) and removes all other list candidates. Then, all list candidates that satisfy predicate `safe_ls_remove` are deleted from set LC . At the end of operation `LIST()`, the reader returns the keys corresponding to list candidates that are still in set LC .

Algorithm of the servers As in Section 5.4, the basic operation of each server i in the LwKVS protocol is to wait for the reception of a message from some client, to perform local computations and to send the corresponding reply back to the client. Messages of type *pre-write*, *write*, *read1* and *read2* correspond to their counterparts from the atomic register implementation. Each of this messages additionally contains the key key used by the high-level operation.

Hence, in the following only the handling of the three new message types *timestamp_request*, *list1* and *list2* are described. When server i receives a message $ts_req\langle key, ts_k \rangle$ sent by client k during a PUT operation, server i retrieves the candidates stored under keys “w” \circ key and “wb” \circ $key \circ *$ from its local KVS and determines the highest timestamp ts corresponding to these candidates. Server i sends timestamp ts in a *timestamp_request_ack* message back to writer k .

Messages of type *list1* and *list2* are sent by the readers during `LIST()` operations to server i . Whenever server i receives a *list1* message $ls1\langle tsl \rangle$ from some reader k , it collects keys from its local KVS in set $AllKeys$. For all keys in $AllKeys$ it determines candidate w from `Get`(“w” \circ key) and adds triples $\langle w.ts, key, w.token \rangle$ to local set LC . For each such candidate w , server i retrieves candidates with higher timestamps than w from `Get`(“wb” \circ $key \circ *$) and adds the corresponding timestamp-key-token triples to set LC . Finally, for all keys “wb” \circ $key \circ ts$ in $AllKeys$, server i adds timestamp-key-token triples from `Get`(“wb” \circ $key \circ ts$) to set LC that have not yet been considered (e.g., if for some key key and timestamp ts , server i has received some candidate in a *read2* message during some `GET(key)` operation without having received this candidate in a *write* message during `PUT(key, *)` operation). Set LC is sent in a *list1_ack* message back to reader k .

Upon reception of a *list2* message $ls2\langle tsl, CLC \rangle$ from reader k , server i resets the set of timestamp-key-flag triples TKF . Next, server i moves through all list candidates lc from received set CLC and calls procedure `VERIFY($lc, lc.key$)`. If the verification is successful (i.e., `VERIFY($lc, lc.key$)` does not return $\langle \perp, \perp \rangle$), then timestamp-key-flag triple $\langle lc.ts, lc, key, flag \rangle$ is added to set TKF . The boolean *flag* is true, if `VERIFY($lc, lc.key$)` returned a pair $\langle lc.ts, \perp \rangle$. Finally, set TKF is sent in a *ls2_ack*(tsl, TV) message back to reader k .

Algorithm 5.9: LWKVS Protocol, GET Operation
(Algorithm of Client k)

Variables and Initialization:

tsr integer, initially 0
 Q set of integers, initially \emptyset
 $TV[1..n]$ vector of sets of structure $\langle ts, value \rangle$, initially \emptyset
 C set of structure $\langle ts, token \rangle$, initially \emptyset
 c_{ret}, c structure $\langle ts, token \rangle$
 v, v_{ret} values

Predicates:

$\text{highCand}(c) \triangleq c \in C : (\forall c' \in C : c.ts \geq c'.ts)$
 $\text{safe}(c) \triangleq |\{i \in Q : \langle c.ts, v \rangle \in TV[i]\}| \geq t + 1$
 $\text{incomplete}(c) \triangleq |\{i \in Q : \langle c.ts, v \rangle \notin TV[i]\}| \geq n - t$

GET(key)

```

5.9.1   $C \leftarrow Q \leftarrow \emptyset$ 
5.9.2   $TV[i] \leftarrow \emptyset, 1 \leq i \leq n$ 
      /* Round 1
5.9.3   $tsr \leftarrow tsr + 1$ 
5.9.4  send  $rd1\langle tsr, key \rangle$  to all servers
5.9.5  repeat
5.9.6    if received  $rd1\_ack\langle tsr, key, C_i \rangle$  from server  $i$  then
5.9.7       $C \leftarrow C \cup C_i$ 
5.9.8  until received  $rd1\_ack\langle tsr, key, * \rangle$  from  $n - t$  servers
      /* Round 2
5.9.9  send  $rd2\langle tsr, key, C \rangle$  to all servers
5.9.10 repeat
5.9.11   if received  $rd2\_ack\langle tsr, key, TV \rangle$  from server  $i$  then
5.9.12      $Q \leftarrow Q \cup \{i\}; TV[i] \leftarrow TV$ 
5.9.13      $C \leftarrow C \setminus \{c \in C : \text{incomplete}(c)\}$ 
5.9.14   until (received  $rd2\_ack\langle tsr, key, * \rangle$  from  $n - t$  servers)  $\wedge$ 
           ( $\exists c \in C : \text{safe}(c) \wedge \text{highCand}(c)$ )
5.9.15   if  $\text{VALUEOF}(c) \neq \perp$  then return  $\text{VALUEOF}(c)$ 
5.9.16   else return fail

VALUEOF( $c$ )
5.9.17   $c_{ret} \leftarrow c : c \in C \wedge \text{safe}(c) \wedge \text{highCand}(c)$ 
5.9.18   $v_{ret} \leftarrow v : |\{i \in Q : \langle c_{ret}.ts, v \rangle \in TV[i]\}| \geq t + 1$ 
5.9.19  return  $v_{ret}$ 

```

Algorithm 5.10: LwKVS Protocol, LIST Operation
 (Algorithm of Client k)

Variables and Initialization:

 tsl integer, initially 0 LQ set of integers, initially \emptyset $flag$ boolean $TKF[1..n]$ vector of sets of structure $\langle ts, key, flag \rangle$, initially \emptyset LC set of structure $\langle ts, key, token \rangle$, initially \emptyset lc structure $\langle ts, key, token \rangle$

Predicates:

 $highCand_ls(lc) \triangleq$ $lc \in LC : (\forall lc' \in LC : lc.ts \geq lc'.ts \vee lc.key \neq lc'.key)$ $safe_ls(lc) \triangleq |\{i \in LQ : \langle lc.ts, lc.key, flag \rangle \in TKF[i]\}| \geq t + 1$ $safe_ls_remove(lc) \triangleq |\{i \in LQ : \langle lc.ts, lc.key, \mathbf{true} \rangle \in TKF[i]\}| \geq t + 1$ $incomplete_ls(lc) \triangleq |\{i \in LQ : \langle lc.ts, lc.key, * \rangle \notin TKF[i]\}| \geq n - t$

LIST()

```

5.10.1   $LC \leftarrow LQ \leftarrow \emptyset$ 
5.10.2   $TV[i] \leftarrow \emptyset, 1 \leq i \leq n$ 
        /* Round 1
5.10.3   $tsl \leftarrow tsl + 1$ 
5.10.4  send  $ls1\langle tsl \rangle$  to all servers
5.10.5  repeat
5.10.6    if received  $ls1\_ack\langle tsl, LC_i \rangle$  from server  $i$  then
5.10.7       $LC \leftarrow LC \cup LC_i$ 
5.10.8  until received  $ls1\_ack\langle tsl, * \rangle$  from  $n - t$  servers
        /* Round 2
5.10.9  send  $ls2\langle tsl, LC \rangle$  to all servers
5.10.10 repeat
5.10.11   if received  $ls2\_ack\langle tsl, TKF \rangle$  from server  $i$  then
5.10.12      $LQ \leftarrow LQ \cup \{i\}; TKF[i] \leftarrow TKF$ 
5.10.13      $LC \leftarrow LC \setminus \{lc \in LC : incomplete\_ls(lc)\}$ 
5.10.14  until (received  $ls2\_ack\langle tsl, * \rangle$  from  $n - t$  servers)  $\wedge$ 
            $\forall lc \in LC : safe\_ls(lc)$ 
5.10.15  forall  $lc \in LC$  do
5.10.16    if  $highCand\_ls(lc)$  then
5.10.17       $LC \leftarrow LC \setminus \{lc' | lc'.key = lc.key\} \cup \{lc\}$ 
5.10.18  forall  $lc \in LC$  do
5.10.19    if  $safe\_ls\_remove(lc)$  then
5.10.20       $LC \leftarrow LC \setminus lc$ 
5.10.21  return  $\{lc.key | lc \in LC\}$ 

```

Algorithm 5.11: LwKVS Protocol, Algorithm of Server i

Variables and Initialization:

local key value store accessible via operations Put, Get, Delete, and List ts
integer, initially 0

C, tmp set of structure $\langle ts, token \rangle$, initially \emptyset

TV set of structure $\langle ts, value \rangle$, initially \emptyset

- 5.11.1 **upon** reception of $ts_req\langle key, ts_k \rangle$ from client k
 - 5.11.2 $ts \leftarrow \max\{(\text{Get}(\text{"w"} \circ key)).ts, \max\{ts \mid \text{Get}(\text{"wb"} \circ key \circ ts) \neq \emptyset\}\}$
 - 5.11.3 send $ts_req_ack\langle key, ts, ts_k \rangle$ to client k
 - 5.11.4 **upon** reception of $pwr\langle key, ts, pw \rangle$ from client k
 - 5.11.5 Put($\text{"pw"} \circ key \circ ts, pw$)
 - 5.11.6 send $pwr_ack\langle key, ts \rangle$ to client k
 - 5.11.7 **upon** reception of $wr\langle key, w \rangle$ from client k
 - 5.11.8 **if** $w.ts > (\text{Get}(\text{"w"} \circ key)).ts$ **then**
 - 5.11.9 Put($\text{"w"} \circ key, w$)
 - 5.11.10 send $wr_ack\langle key, w.ts \rangle$ to client k
 - 5.11.11 **upon** reception of $rd1\langle tsr, key \rangle$ from client k
 - 5.11.12 $C \leftarrow \emptyset$
 - 5.11.13 $C \leftarrow C \cup \text{Get}(\text{"w"} \circ key)$
 - 5.11.14 **forall** $ts' > (\text{Get}(\text{"w"} \circ key)).ts$ s.t. $\text{Get}(\text{"wb"} \circ key \circ ts') \neq \emptyset$ **do**
 - 5.11.15 $C \leftarrow C \cup \text{Get}(\text{"wb"} \circ key \circ ts')$
 - 5.11.16 send $rd1_ack\langle tsr, key, C \rangle$ to client k
 - 5.11.17 **upon** reception of $rd2\langle tsr, key, CC \rangle$ from client k
 - 5.11.18 **for** $c \in CC$ **do**
 - 5.11.19 $tmp \leftarrow \text{Get}(\text{"wb"} \circ key \circ c.ts)$
 - 5.11.20 Put($\text{"wb"} \circ key \circ c.ts, tmp \cup \{c\}$)
 - 5.11.21 $TV \leftarrow TV \cup \text{VERIFY}(c, key)$
 - 5.11.22 send $rd2_ack\langle tsr, key, TV \rangle$ to client k
-

Algorithm 5.12: LwKVS Protocol, Algorithm of Server i , cont.

Additional Variables:
AllKeys set of keys, initially \emptyset
LC set of structure $\langle ts, key, token \rangle$, initially \emptyset
w structure $\langle ts, token \rangle$, initially $\langle 0, token_0 \rangle$
wb set of structure $\langle ts, token \rangle$
TKF set of structure $\langle ts, key, flag \rangle$, initially \emptyset
tval structure $\langle ts, value \rangle$

5.12.1 **upon** reception of $ls1\langle tsl \rangle$ from client k
5.12.2 $LC \leftarrow \emptyset$
5.12.3 $AllKeys \leftarrow \text{List}()$
5.12.4 **forall** key s.t. “w” \circ $key \in AllKeys$ **do**
5.12.5 $w \leftarrow \text{Get}(\text{“w”} \circ key)$
5.12.6 $LC \leftarrow LC \cup \{ \langle w.ts, key, w.token \rangle \}$
5.12.7 **forall** $ts' > (\text{Get}(\text{“w”} \circ key)).ts$ s.t. $\text{Get}(\text{“wb”} \circ key \circ ts') \neq \emptyset$ **do**
5.12.8 $wb \leftarrow \text{Get}(\text{“wb”} \circ key \circ ts')$
5.12.9 $LC \leftarrow LC \cup \{ \langle ts', key, lc.token \rangle | lc \in wb \}$
5.12.10 **forall** ts, key s.t. “wb” \circ $key \circ ts \in AllKeys$ **do**
5.12.11 **if** $ts > (\text{Get}(\text{“w”} \circ key)).ts \wedge \text{Get}(\text{“wb”} \circ key \circ ts) \neq \emptyset$ **then**
5.12.12 $wb \leftarrow \text{Get}(\text{“wb”} \circ key \circ ts)$
5.12.13 $LC \leftarrow LC \cup \{ \langle ts, key, lc.token \rangle | lc \in wb \}$
5.12.14 send $ls1_ack\langle tsl, LC \rangle$ to client k
5.12.15 **upon** reception of $ls2\langle tsl, CLC \rangle$ from client k
5.12.16 $TKF \leftarrow \emptyset$
5.12.17 **forall** $lc \in CLC$ **do**
5.12.18 $tval \leftarrow \text{VERIFY}(lc, lc.key)$
5.12.19 **if** $tval \neq (\perp, \perp)$ **then**
5.12.20 **if** $tval.val \neq \perp$ **then** $TKF \leftarrow TKF \cup \{ \langle lc.ts, lc.key, \text{false} \rangle \}$
5.12.21 **else** $TKF \leftarrow TKF \cup \{ \langle lc.ts, lc.key, \text{true} \rangle \}$
5.12.22 send $ls2_ack\langle tsl, TKF \rangle$ to client k

Algorithm 5.13: VERIFY Procedure with Hash Function (H)

$\text{VERIFY}(c, key)$

5.13.1 $pw \leftarrow \text{Get}(\text{“pw”} \circ key \circ c.ts)$
5.13.2 **if** $pw.commitment = H(c.token)$ **then**
5.13.3 **return** $\langle c.ts, pw.value \rangle$
5.13.4 **else**
5.13.5 **return** $\langle \perp, \perp \rangle$

Algorithm 5.14: VERIFY Procedure with Secret Sharing (S)

```

VERIFY( $c, key$ )
5.14.1    $pw \leftarrow \text{Get}("pw" \circ key \circ c.ts)$ 
5.14.2   if  $(x, y) = pw.commitment \wedge$ 
          $(a_0, \dots, a_t) = c.token \wedge$ 
          $y = a_0 + xa_1 + x^2a_2 + \dots + x^ta_t$  then
5.14.3     return  $\langle c.ts, pw.value \rangle$ 
5.14.4   else
5.14.5     return  $\langle \perp, \perp \rangle$ 

```

5.5.3 Definition of Regular List Operations

As discussed in Section 5.5.1, the introduced LwKVS protocol constitutes a Byzantine fault-tolerant implementation of a lightweight and robust KVS providing *atomic* PUT, GET, and DELETE operations and *regular* LIST operations. Operations are defined to be *atomic* if their invocation and response events can be arranged in a *linearizable* order for any execution of the LwKVS protocol. However, there exists no standard definition for protocols that provides *atomic* as well as *regular* operations. Hence, in the following a definition of a *regular* LIST operation is given in the context of *atomic* PUT, GET, and DELETE operations.

Let σ be a history of an execution of the LwKVS protocol implementing a key-value store. Then, let $put_get_delete(\sigma)$ denote the subsequence of history σ containing only invocation and response events of PUT, GET, and DELETE operations in σ . Operations PUT, GET, and DELETE are called *atomic*, if for any history σ , sequence $put_get_delete(\sigma)$ is linearizable with respect to the implemented functionality of a KVS.

To define *regular* semantics of LIST operations, the notion of active and passive keys is introduced. Intuitively, a key key is *active* if the implemented KVS stores a non- \perp value under key . The next definition also defines operations that activate and deactivate keys.

Definition 5.8. Let σ be a history such that $put_get_delete(\sigma)$ is linearizable. Let π_{pgd} be the sequential permutation of the completion of $put_get_delete(\sigma)$ as given in Definition 2.2 on page 25.

Initially all keys are defined to be *deactivated*. The first PUT operation in π_{pgd} accessing key k is said to *activate* k . We define recursively: The first DELETE operation in π_{pgd} accessing key k after an operation that activates k is said to *deactivate* k . The first PUT operation in π_{pgd} accessing k after an operation that deactivates k is said to *activate* k . Any operation that *activates* or *deactivates* key k is called a *k-changing* operation.

The following definition introduces the notion of *list regularity* for LIST opera-

tions. Finally, a LIST operation provided by a KVS is called *regular*, if any history σ is *list-regular*.

Definition 5.9 (List-Regularity). A history σ is *list-regular* if for each LIST operation l in σ and each key k the following conditions are satisfied:

1. $put_get_delete(\sigma)$ is linearizable.
2. $prefix_{\rightarrow l}(\pi_{pgd})$ is the largest prefix of π_{pgd} (see Definition 5.8) such that the last operation in π_{pgd} precedes l in σ , and
3. if the last k -changing operation in $prefix_{\rightarrow l}(\pi_{pgd})$ activates (deactivates) k , and l is not concurrent with any other operation deactivating (activating) k , then k is contained (is not contained) in the set of keys returned by l , and
4. if the last k -changing operation in $prefix_{\rightarrow l}(\pi_{pgd})$ activates (deactivates) k , and l is concurrent with any other operation deactivating (activating) k , then k may or may not be contained in the set of keys returned by l .
5. if there is no k -changing operation in $prefix_{\rightarrow l}(\pi_{pgd})$, and l is not concurrent with any other operation activating k , then k is not contained in the set of keys returned by l , and
6. if there is no k -changing operation in $prefix_{\rightarrow l}(\pi_{pgd})$, and l is concurrent with any other operation activating k , then k may or may not be contained in the set of keys returned by l .

5.5.4 Proof of Correctness of the LwKVS Protocol

We show in this section that the LwKVS protocol given as Algorithms 5.8, 5.2, 5.3, 5.9, 5.10, 5.11, 5.12, 5.13, and 5.14 implements a key-value store with atomic PUT, GET, and DELETE operations and regular LIST operation.

We first show that the operations implemented by the LwKVS protocol satisfy the safety properties.

Corollary 5.10. Let $PUT(key, v)$ be a put operation of client k that stores value v with timestamp ts_v and key key to the key-value storage. Let $PUT(key, v')$ be a put operation of client k' such that $PUT(key, v)$ precedes $PUT(key, v')$. Then, client k does not assign a timestamp $\leq ts_v$ to value v' (in line 5.8.6).

Proof. By the assumption that $PUT(key, v)$ precedes $PUT(key, v')$, client k sends during the write phase of operation $PUT(key, v)$ a $wr\langle key, w \rangle$ message to all servers (line 5.8.15) where $w.ts = ts_v$ and $w.value = v$ (line 5.8.13). As correct servers update entry with key “w” \circ key in their local KVS only on receiving a value

with a higher timestamp (line 5.11.8), at the end of operation $\text{PUT}(key, v)$, a call of $\text{Get}("w" \circ key)$ on $t + 1$ correct servers returns a value w with corresponding timestamp $w.ts \geq ts_v$. Hence, at least one of these correct servers will reply to the $ts_req\langle key, * \rangle$ message during operation $\text{PUT}(key, v')$ of client k' with a timestamp $\geq ts_v$ (in such a *timestamp_request_ack* message the maximum of $(\text{Get}("w" \circ key)).ts$ and timestamps ts' such that $\text{Get}("wb" \circ key \circ ts') \neq \emptyset$ is sent; lines 5.11.1–5.11.3). Thus, by lines 5.8.4–5.8.5, when client k' has received *timestamp_request_ack* messages from $n - t$ servers, its variable ts is not smaller than ts_v . By line 5.8.6, the timestamp assigned to v' is greater than ts_v : The timestamps that are assigned during PUT operation is constructed as $n_w \cdot i + k$ where n_w is the number of writers, i is a counter value that is incremented and k is the client id . Let $ts' = n_w \cdot i + j$ be a timestamp that is going to be incremented by client k' . Then the new timestamp of client k' is computed as $n_w \cdot (i + 1) + k'$. \square

Corollary 5.11. Let $\text{GET}(key)$ be a get operation of *correct* client k that retrieves value v with timestamp ts_v and key key from the key-value storage. Let $\text{PUT}(key, v')$ be an operation of client k' such that $\text{GET}(key)$ precedes $\text{PUT}(key, v')$. Then, client k does not assign a timestamp $\leq ts_v$ to value v' .

Proof. By assumption, client k has returned value v (with timestamp ts_v). Hence, for client k there has been a candidate c in the set of candidates C where $c.ts = ts_v$ (line 5.9.14). Set C containing candidate c is sent during second round of operation $\text{GET}(key)$ in a *read2* message to $n - t$ servers (line 5.9.9), and therefore received by $t + 1$ correct servers. After having received such a *read2* message, on these $t + 1$ correct servers holds $(\text{Get}("w" \circ key)).ts \geq ts_v$ or $\text{Get}("wb" \circ key \circ ts') \neq \emptyset$ for some $ts' \geq ts_v$: If $ts_v = c.ts \leq \text{Get}("w" \circ key).ts$, the claim is true. Else, $\text{Put}("wb" \circ key, ts')$ is called where $ts' \geq c.ts = ts_v$ (line 5.11.20), and the claim is also true.

Hence, at least one of these correct servers will reply to the $ts_req\langle key, * \rangle$ message during operation $\text{PUT}(key, v')$ of client k' with a timestamp $\geq ts_v$ (in such a *timestamp_request_ack* message the maximum of $(\text{Get}("w" \circ key)).ts$ and timestamps ts' such that $\text{Get}("wb" \circ key \circ ts') \neq \emptyset$ is sent; lines 5.11.1–5.11.3). Hence, when client k' has received *timestamp_request_ack* messages from $n - t$ servers, its variable ts is not smaller than ts_v . Analogously, to the proof of Corollary 5.10, it directly follows that client k' does not assign a timestamp $\leq ts_v$ to value v' . \square

The next Corollary proves that PUT and GET operations satisfy *regularity*.

Corollary 5.12. Let $\text{PUT}(key, v)$ be a put operation of client k that stores value v with timestamp ts_v and key key to the key-value storage. Let $\text{GET}(key)$ be a get operation of client k' such that $\text{PUT}(key, v)$ precedes $\text{GET}(key)$. Then, operation $\text{GET}(key)$ does not return a value with timestamp $< ts_v$.

Proof. In the first round of operation $\text{GET}(key)$ client k' receives candidates from “w” \circ key entries (lines 5.11.13 and 5.11.16) of the local KVS of $2t + 1$ servers. Client k' collect these candidates in set C (line 5.9.7). After the first round of $\text{GET}(key)$, let w' be the candidate with the highest timestamp in set C at client k' , which has been sent by a correct server (from its “w” \circ key entry of the local KVS; line 5.11.13). We first show that $w'.ts \geq ts_v$:

As operation $\text{PUT}(key, v)$ is complete, client k has sent w_v (together with key) where $w_v.ts = ts_v$ to $2t + 1$ servers during the write-phase (line 5.8.15). Hence, at least $t + 1$ correct servers have received this $wr\langle key, w_v \rangle$ message from client k . For these servers holds upon the reception of the *write* message: Either $w_v.ts \leq \text{Get}(\text{“w”} \circ key).ts$. In this case some other candidate w' has been received where $w'.ts = (\text{Get}(\text{“w”} \circ key)).ts \geq w_v.ts$. Or, $w_v.ts > \text{Get}(\text{“ts”} \circ key).ts$. Then, candidate w_v is stored to the local KVS by call $\text{Get}(\text{“w”} \circ key, w_v)$ (line 5.11.9). Thus, at least one correct server adds w_v or a candidate with higher timestamp to set C (line 5.11.13) and sends it as a reply to *read1* message of $\text{GET}(key)$ to client k' (line 5.11.16). This means that client k' adds w_v (or a candidate with higher timestamp) to set C (line 5.9.7). Thus, the claim $w'.ts \geq ts_v$ is true.

Next, we show that during operation $\text{GET}(key)$ client k' never removes candidate w' from set C . Let $\text{PUT}(key, v')$ be the operation storing candidate w' to the KVS, i.e., let v' be the value written by operation $\text{PUT}(key, v')$ with timestamp $w'.ts \geq ts_v$. We continue the proof separately for the two variants (H) and (S):

Hash Function, Case (H). In the pre-write-phase of $\text{PUT}(key, v')$, the client has sent $\langle key, w'.ts, (H(w'.token), v') \rangle$ to $2t + 1$ servers (line 5.8.11), where $w'.token$ is the output of function $\text{GETRANDOM}()$ when called by the client during operation $\text{PUT}(key, v')$ (line 5.3.1). As w' is stored in the “w” \circ key entry of the local KVS of some correct server, the pre-write phase of $\text{PUT}(key, v')$ is complete. Hence, at least $t + 1$ correct servers hold $\langle H(w'.token), v' \rangle$ in the “pw” \circ $key \circ w'.ts$ entry of their local KVS (line 5.11.5). Thus, call of $\text{VERIFY}(w', key)$ (line 5.11.21) at any of these $t + 1$ correct servers would return $\langle w'.ts, v' \rangle$ (line 5.14.3). This implies that there are at most $2t$ servers that reply with timestamp-value pairs different from $\langle w'.ts, v' \rangle$ to *read2* message of client k' .

Secret Sharing, Case (S). In the pre-write-phase of $\text{PUT}(key, v')$, the client has sent $\langle key, w'.ts, (x_i, a'_0 + x_i a'_1 + x_i^2 a'_2 + \dots + x_i^t a'_t), v' \rangle$ to a set of $2t + 1$ servers i (line 5.8.11), where a'_0, a'_1, \dots, a'_t and x_i is the output of $\text{GETRANDOM}()$ when called during $\text{PUT}(key, v')$ (lines 5.2.2 and 5.2.3). Note that $w'.token = (a'_0, a'_1, \dots, a'_t)$. As w' is stored in the “w” \circ key entry of the local KVS of some correct server, the pre-write phase of $\text{PUT}(key, v')$ is complete. Hence, at least $t + 1$ correct servers i hold $\langle key, w'.ts, (x_i, a'_0 + x_i a'_1 + x_i^2 a'_2 + \dots + x_i^t a'_t), v' \rangle$ in the “pw” \circ $key \circ w'.ts$ entry of their local KVS (line 5.11.5). Thus, call of $\text{VERIFY}(w', key)$ (line 5.11.21) at any of these $t + 1$ correct servers would return $\langle w'.ts, v' \rangle$ (line 5.13.5). This implies that there are at most $2t$ servers that reply with timestamp-value pairs different

from $\langle w'.ts, v' \rangle$ to *read2* message of client k' .

This implies that in both algorithm variants w' is never removed from the set of candidates C by client k' (line 5.9.13). Hence, predicate *highCand* makes sure that $\text{GET}(key)$ does not return a value with timestamp $< ts_v$. \square

The next Corollary proves that PUT and GET operations satisfy the additional properties of *atomicity*.

Corollary 5.13. Let $\text{GET}'(key)$ be an operation of *correct* client k' that has returned value v (with timestamp ts_v) from the key-value storage. Let $\text{GET}(key)$ be an operation of client k such that $\text{GET}'(key)$ precedes $\text{GET}(key)$. Then, operation $\text{GET}(key)$ does not return a value with timestamp $< ts_v$.

Proof. As $\text{GET}'(key)$ has returned v there has been a candidate w_v during the second round of $\text{GET}'(key)$ that satisfied *safe*(w_v) and *highCand*(w_v) and where $w_v.ts = ts_v$. We first show that w_v has been written in the write-phase of some put operation $\text{PUT}(key, v)$ and that the corresponding pre-write phase is complete. We proceed the proof separately for the two variants (H) and (S):

Hash Function, Case (H). By the definition of *safe*(w_v) (in Algorithm 5.9), there exist $t + 1$ servers that have replied with *read2_ack* messages (line 5.11.22) containing $\langle ts_v, v \rangle$ to client k' during $\text{GET}'(key)$. Hence, there is at least one correct server where call $\text{VERIFY}(w_v, key)$ returned $\langle ts_v, v \rangle$ which implies that one correct server holds pw in its “pw” $\circ key \circ w_v.ts$ entry of the local KVS such that $H(w_v.token) = pw.commitment$. By the hiding property of hash function H , any server can know $w_v.token$ only after it has been sent (line 5.8.15) during the write-phase of some put operation $\text{PUT}(key, v)$ which further implies that the corresponding pre-write-phase is complete.

Secret Sharing, Case (S). By the definition of *safe*(w_v), there exist $t + 1$ servers that have replied with *read2_ack* messages (line 5.11.22) containing $\langle ts_v, v \rangle$ to client k' during $\text{GET}'(key)$. Hence, there is at least one correct server where call $\text{VERIFY}(w_v, key)$ returned $\langle ts_v, v \rangle$ which implies that one correct server i holds pw in its “pw” $\circ key \circ w_v.ts$ entry of the local KVS such that $(x_i, y_i) = pw.commitment$, $(a_0, \dots, a_t) = w_v.token$, and $a_0 + x_i a_1 + x_i^2 a_2 + \dots + x_i^t a_t = y_i$. By the hiding property of the secret sharing scheme, any server can know $w_v.token$ only after it has been sent during the write-phase some put operation $\text{PUT}(key, v)$ which further implies that the corresponding pre-write-phase is complete.

Note, that in both cases, at the point in time when client k' returns value v at the end of operation $\text{GET}'(key)$, the pre-write phase of $\text{PUT}(key, v)$ is complete. Next, we show that during first round of $\text{GET}(key)$ client k adds a candidate with timestamp $\geq ts_v$ to the set of candidates C which is never removed:

As v has been returned by $\text{GET}'(key)$, w_v must have been a candidate for client k' , i.e., $w_v \in C$ during the first round of $\text{GET}'(key)$ at client k' . As operation

$\text{GET}'(\text{key})$ is complete, set C containing w_v has been sent to $2t + 1$ servers in read2 messages. At least $t + 1$ correct servers receive w_v in such a read2 message and for each such server i holds: If $w_v.ts \leq \text{Get}(\text{"w"} \circ \text{key}).ts$ server i already holds a candidate with timestamp $ts' \geq w_v.ts$ in the $\text{"w"} \circ \text{key}$ entry of its local KVS (this implies that there exist some put operation that stored candidate w' , $w'.ts = ts'$ during write phase to the KVS of server i). Else, w_v is added by server i to $\text{"wb"} \circ \text{key} \circ w_v.ts$ entry of its local KVS (line 5.11.20).

As $\text{GET}'(\text{key})$ precedes $\text{GET}(\text{key})$, client k will receive set of candidates C_i containing candidate w_v or w' (if it exists) from at least one correct server i during first round in a read1_ack message. This is because server i either adds w_v from its $\text{"wb"} \circ \text{key} \circ w_v.ts$ entry of its local KVS to set C_i (line 5.11.15) or a candidate w' with higher timestamp than $w_v.ts$ (line 5.11.13). As the corresponding pre-write phases of w_v and w' (if it exists) are complete, by the same arguments as in the proof of Corollary 5.12, w_v or w' (if it exists) is never removed from set of candidates by client k during $\text{GET}(\text{key})$. Thus by predicate highCand , $\text{GET}(\text{key})$ does not return a value with timestamp $< ts_v$. \square

The next two corollaries prove that only values are returned that actually have been written by some PUT operation.

Corollary 5.14. No $\text{GET}(\text{key})$ operation of a *correct* client k returns a value with timestamp < 0 .

Proof. Initially, if no value has been written to the KVS, all correct servers return value $\langle 0, \text{key}_0, \text{token}_0 \rangle$ on call $\text{Get}(\text{"w"} \circ \text{key} \circ 0)$ and value $\langle \text{commitment}_0, \perp \rangle$ on call $\text{Get}(\text{"pw"} \circ \text{key} \circ 0)$. As client k is correct and increases its timestamp at the beginning of a PUT operation (line 5.8.6), entry $\text{"w"} \circ \text{key} \circ 0$ is only updated with candidates with higher timestamps. Hence, for any operation $\text{GET}()$ holds, that any correct server only proposes candidates with timestamp ≥ 0 . By the same arguments as in the proof of Corollary 5.1, such a candidate of a correct server is never removed from set of candidates by client k during $\text{GET}(\text{key})$. Thus by predicate highCand , $\text{GET}(\text{key})$ does not return a value with timestamp < 0 . \square

Corollary 5.15. Let $\text{GET}(\text{key})$ be a get operation of *correct* client k that returns value $v \neq \perp$ (with timestamp ts_v). Then, there exists an operation $\text{PUT}(\text{key}, v)$ such that $\text{GET}(\text{key})$ does not precede $\text{PUT}(\text{key}, v)$. If $\text{GET}(\text{key})$ returns \perp (with timestamp 0), then there exists no $\text{PUT}(\text{key}, v)$ operation that precedes $\text{GET}(\text{key})$.

Proof. If $\text{GET}(\text{key})$ returns value v , there exists a candidate $c \in C$ for client k such that $\text{safe}(c)$ holds (line 5.9.14). This implies that there are $t + 1$ servers at which call $\text{VERIFY}(c, \text{key})$ returns timestamp-value pair $\langle ts_v, v \rangle$, $c.ts = ts_v$, which is sent in read2_ack messages to client k . As there are at most t malicious servers, among

these servers at least one server is correct. By line 5.11.21, the correct server has determined v from $\text{Get}(\text{"pw"} \circ \text{key} \circ c.ts).value$. As $\text{Get}(\text{"pw"} \circ \text{key} \circ c.ts).value \neq \perp$, the correct server has received a *pre-write* message of some $\text{PUT}(key, v)$ operation containing pw such that $pw.value = v$. This *pre-write* message was received by the correct server *before* it received the *read2* message during $\text{GET}(key)$. Thus, $\text{GET}(key)$ does not precede $\text{PUT}(key, v)$.

If $\text{GET}(key)$ returns value \perp , there exists a candidate $c_0 \in C$, $c_0.ts = 0$ for client k such that $\text{safe}(c_0)$ and $\text{highCand}(c_0)$ holds (line 5.9.14) and $\text{VALUEOF}(c_0)$ returns \perp . This implies that for all candidates $c \in C$ such that $c.ts > 0 = c_0.ts$ holds $\text{incomplete}(c)$ and they have been removed from C (line 5.9.13). Let $\text{PUT}(key, v)$ be the operation that has written such a candidate c during write-phase.

Predicate $\text{incomplete}(c)$ implies that $2t + 1$ servers have replied with *read2_ack* messages not containing any timestamp-value pairs $\langle c.ts, v \rangle$. Hence, at $t+1$ correct servers, $\text{Get}(\text{"pw"} \circ \text{key} \circ c.ts)$ does not return $\langle \text{commitment}_v, v \rangle$. This further implies that $t + 1$ correct servers did not receive a $\text{pwr}\langle key, ts_v, \langle \text{commitment}_v, v \rangle \rangle$ message before they reply with *read2_ack* messages. Thus, the pre-write phase of $\text{PUT}(key, v)$ is not complete when second round of $\text{GET}(key)$ is initiated, and therefore, $\text{PUT}(key, v)$ does not precede $\text{GET}(key)$. \square

The next two corollaries show that $\text{LIST}()$ operation satisfies *regularity*.

Corollary 5.16. Let $\text{PUT}(key, v)$ be a put operation of client k and let $\text{LIST}()$ be a list operation of client k' . If there is no $\text{DELETE}(key)$ operation succeeding $\text{PUT}(key, v)$ (that is initiated before $\text{LIST}()$ finishes), then key is contained in set Keys returned by $\text{LIST}()$.

Proof. By the correctness of the PUT operation, let us assume that client k has sent $pw_v = \langle \text{commitment}_v, v \rangle$ in the pre-write phase and $w_v = \langle ts_v, key, \text{token}_v \rangle$ in the write-phase of operation $\text{PUT}(key, v)$ to the servers. Variables pw_v and w_v have been received by at least $t+1$ correct servers. As $\text{PUT}(key, v)$ precedes $\text{LIST}()$, at least one of the $t + 1$ correct servers receives a *list1* message (line 5.10.4) from client k' after having received w_v from client k . Upon reception of a *list1* message, the correct server collects all used keys by locally calling $\text{List}()$ in set AllKeys (line 5.12.3). For each local key of type $\text{"w"} \circ key$, the server retrieves for key the candidate (with highest timestamp) that has been stored in some write phase (line 5.12.5). The corresponding candidate w is collected as triple $\langle w.ts, key, w.token \rangle$ in set LC (line 5.12.6). As $\text{PUT}(key, v)$ precedes $\text{LIST}()$, set LC contains a triple $\langle ts', key, \text{token} \rangle$ where $ts' \geq ts_v$. Hence, after client k' has received LC in a *list1_ack* message, it adds lc_v such that $lc_v.ts \geq ts_v$ and $lc_v.key = key$ to its local set LC (lines 5.12.14 and 5.10.7).

As candidate w_v has been written to a correct server, $t + 1$ correct servers have stored the corresponding pre-write information pw_v . Hence, in the second round

of $\text{LIST}()$, $t + 1$ correct servers reply with sets containing $\langle ts_v, key, flag \rangle$ triple such that lc_v turns **safe_ls** at client k' (line 5.10.14). As no $\text{DELETE}(key)$ operation happens after $\text{PUT}(key, v)$, variable $flag$ is **false** and **safe_ls_remove** does not hold for lc_v or a candidate lc' where $lc'.key = key$ and $lc'.ts \geq lc_v.ts$. This implies that client k' never removes candidate for key from set LC (lines 5.10.17 and 5.10.20) and key is contained in the set of returned keys (line 5.10.21). \square

Corollary 5.17. Let $\text{GET}(key)$ be a get operation of client k that returns value v (with timestamp ts_v) and let $\text{LIST}()$ be an operation of client k' . If there is no $\text{DELETE}(key)$ operation succeeding $\text{GET}(key)$ (that is initiated before $\text{LIST}()$ finishes), then key is contained in set $Keys$ returned by $\text{LIST}()$.

Proof. The proof is very similar to the proof of Corollary 5.16. Hence, we only show that client k' adds candidate lc_v , such that $lc_v.key = key$ to set LC which is never removed.

By the implementation of $\text{GET}(key)$ operation, client k writes back some candidate w_v during second phase corresponding to value v . Hence, at $t + 1$ correct servers entry “wb” $\circ key \circ ts_v$ in the local KVS contains w_v or “w” $\circ key$ holds a candidate with a higher timestamp $ts' \geq ts_v$. Hence, upon reception of a *list1* message from client k' during $\text{LIST}()$, a correct server adds a triple $\langle ts', key, w'.token \rangle$ to set LC : Here, $ts' \geq ts_v$ and w' is a candidate, that either has been written in the write-phase of some PUT operation (line 5.12.6), or during the second round of some GET operation, where no candidate (line 5.12.13) or no candidate with a higher timestamp (line 5.12.9) has been written for key . When client k' receives set LC in a *list1_ack* message from a correct server, it adds lc_v such that $lc_v.ts \geq ts_v$ and $lc_v.key = key$ to set LC (lines 5.12.14 and 5.10.7).

The remainder of the proof is analogous to the proof of Corollary 5.16. Hence, key is contained in the set of returned keys. \square

Lemma 5.18. The LwKVS protocol given as Algorithms 5.8, 5.2, 5.3, 5.9, 5.10, 5.11, 5.12, 5.13, and 5.14 implements a key-value store where PUT , GET , and DELETE operations are *atomic* and LIST operations are *regular*.

Proof. Correctness follows directly from Corollaries 5.10, 5.11, 5.12, 5.13, 5.14, 5.15, 5.16, and 5.17. \square

We proceed by showing that none of the operations implemented by the LwKVS protocol blocks.

Lemma 5.19. The GET and PUT operations implemented by the LwKVS protocol are *wait-free*.

Proof. By assumption, there are at most t malicious servers. As the client never waits for more than $n - t$ replies during read-timestamp, pre-write and write phase of a PUT operation and during the first round of a GET operation, it remains to show that the algorithm does also not block during the second round of any GET operation.

We assume for contradiction that some operation $\text{GET}(key)$ of client k blocks during the second round, i.e., we consider the point in time where all correct servers have replied with *read2_ack* messages (line 5.9.14). We first show, that the set of candidates C is not empty. Therefore, we assume that w is the last candidate that has been written in line 5.11.9 (during write phase of some $\text{PUT}(key, v_w)$ operation) to some correct server i (or the initial candidate $\langle 0, key_0, token_0 \rangle$ if no candidates have been written yet) before server i replies to *read1* message during $\text{GET}(key)$ (line 5.11.16). Hence, w is contained in C_i received from server i during first round of $\text{GET}(key)$ and w is added to C by client k (line 5.9.7). As the corresponding pre-write phase precedes $\text{GET}(key)$, w is never removed from set C (cf. proof of Corollary 5.12).

We proceed by showing that all candidates $c \in C$ are **safe**. For contradiction, we assume that there exists a candidate c contained in set C which is not **safe**. Candidate c has been sent by some servers in a *read1_ack* message to client k during $\text{GET}(key)$ (line 5.11.16). At the end of the first round of $\text{GET}(key)$, there exists (Case 1) either some $\text{PUT}(key, *)$ operation that has stored c during write-phase to some server, (Case 2) or there is no such operation as c has been fabricated by some malicious server.

Case 1 There exists a $\text{PUT}(key, *)$ operation that has stored candidate c during write-phase onto some server. Hence, the corresponding pre-write phase has been completed before the second round of $\text{GET}(key)$ is started. It directly follows that c is **safe**, as on $t + 1$ correct servers $\text{VERIFY}(c, key)$ returns $\langle c.ts, v \rangle$, which is a contradiction.

Case 2 As candidate c has not been written at all, by the hiding property of hash function and the secret sharing scheme, on $2t + 1$ correct servers $\text{VERIFY}(c, key)$ does not return $\langle c.ts, v \rangle$. Thus, c is removed from set C (line 5.9.13).

Therefore, candidates in C are either **safe** or eventually removed from C . Thus, no $\text{GET}(key)$ operation blocks in the second round (line 5.9.14) and implemented PUT and GET operations are *wait-free*. \square

Lemma 5.20. The DELETE and LIST operations implemented by the LWKVS protocol are *wait-free*.

Proof. By the proof of Lemma 5.19 it directly follows that DELETE operations do not block, as during DELETE operation only a wait-free PUT operation is called. The first round of LIST operation does not block as the client never waits for more

then $n - t$ replies. Thus, it remains to show that the algorithm does not block during the second round of any LIST operation.

We assume for contradiction that some operation LIST() of client k blocks during the second round, i.e., we consider the point in time where all correct servers have replied with *list2_ack* messages (line 5.10.14). We first show, that the set of candidates LC is not empty. Therefore, we assume that w is the last candidate that has been written in line 5.11.9 (during write phase of some PUT(key, v_w) operation) to some correct server i (or the initial candidate $\langle 0, key_0, token_0 \rangle$ if no candidates have been written yet) before server i replies to *list1* message during LIST() (line 5.12.14). Hence, the triple $\langle w.ts, key, w.token \rangle = lc_w$ is contained in LC_i received from server i during first round of LIST() and lc_w is added to LC by client k (line 5.10.7). As the corresponding pre-write phase precedes LIST(), lc_w is never removed from set LC (cf. proof of Corollary 5.17).

We proceed by showing that all candidates $lc \in LC$ are *safe_ls*. For contradiction, we assume that there exists a candidate lc contained in set LC which is not *safe_ls*. Candidate lc has been sent by some servers in a *list1_ack* message to client k during LIST() (line 5.12.14). At the end of the first round of LIST(), there exists (Case 1) either some PUT($key, *$) or DELETE(key) operation that has stored lc during write-phase to some server, (Case 2) or there is no such operation as lc has been fabricated by some malicious server.

Case 1 There exists a PUT($key, *$) or DELETE(key) operation that has stored candidate lc during write-phase onto some server. Hence, the corresponding pre-write phase has been completed before the second round of LIST() is started. It directly follows that lc is either *safe_ls* (or *safe_ls.remove*), as on $t + 1$ correct servers VERIFY(lc, key) returns $\langle c.ts, v \rangle$ (or $\langle c.ts, \perp \rangle$, respectively), which is a contradiction.

Case 2 As candidate lc has not been written at all, by the hiding property of hash function and the secret sharing scheme, on $2t + 1$ correct servers VERIFY(lc, key) does not return $\langle lc.ts, * \rangle$. Thus, lc is removed from set LC (line 5.10.13).

Therefore, candidates in LC are either *safe_ls* or eventually removed from LC . Thus, no LIST() operation blocks in the second round (line 5.10.14) and implemented DELETE and LIST operations are *wait-free*. \square

Theorem 5.21. The LwKVS protocol given as Algorithms 5.8, 5.2, 5.3, 5.9, 5.10, 5.11, 5.12, 5.13, and 5.14 implements a key-value store with atomic PUT, GET, and DELETE operations and regular LIST operations in a wait-free manner.

Proof. The correctness follows directly from Lemmas 5.18, 5.19, and 5.20. \square

5.6 Conclusion & Discussion

This chapter has introduced as the two final contributions (C5) and (C6) of this thesis the protocols LwR and LwKVS implementing an atomic register and a key-value store, respectively. Both protocols are robust and Byzantine fault-tolerant implementations, i.e., the implemented operations are wait-free and the optimal number of server replicas of $3t+1$ is required to tolerate t Byzantine faulty servers. Moreover, the focus of both protocols is on being lightweight: The worst-case latency in the number of communication rounds between clients and servers is minimal, no expensive techniques to ensure self-verifying data are used, the protocols are scalable in the number of clients and tolerate any number of malicious readers.

Known results from existing works demonstrate a fundamental difference between robust storage implementations that use self-verifying data (authenticated data model) and those implementations where data are non-authenticated. In the authenticated data model it is possible to atomically read in two communication rounds and to write in a single round, if multiple writers are allowed in two rounds [DGLV05]. However, with non-authenticated data, a worst-case latency of four communication rounds for atomic reads has shown to be optimal for scalable implementations, and two (three) rounds for write operations in the single (multiple) writer case [DGM⁺11]. The LwR protocol proposed in this chapter although using non-authenticated data implements an atomic register where READ and WRITE operations complete after two communication rounds, and thereby closes this fundamental gap between implementations in the authenticated and non-authenticated data model to a minimum. Especially, the LwR protocol achieves the same worst-case read latency of two communication rounds that is also optimal for robust implementations in the authenticated data model [DGLV05]. Moreover, by employing a *commitment scheme*, both protocols LwR and LwKVS also tolerate an unbounded number of malicious readers and thereby constitute the first implementations in the non-authenticated data model providing this vital feature.

The LwKVS protocol introduced as contribution (C6) of this thesis implements a robust and Byzantine fault-tolerant key-value store. The provided operations are based on the atomic READ and WRITE operations of the LwR protocol and are lightweight in the same sense. A difference is that the PUT and DELETE operations of the LwKVS protocol require three communication rounds as multiple writers are supported.

The proposed LwKVS leaves room for several possible optimizations. The implemented LIST operation achieves only regular semantics in order to perform as few as two communication rounds. Hence, the thesis at hand leaves for future work the conjecture that atomic, but obstruction-free LIST operations can be implemented: The idea is, as in protocols taking a snapshot [Fic05], to read in

“waves” from all keys. If two sequential “waves” read the same data, the LIST operation can be linearized with respect to all other operations. However, to achieve two consistent “waves”, the LIST operation has to execute in isolation.

In a typical system setting, the servers are hosted within the same data center and clients send their requests from outside. This means that communication within the data center is orders of magnitude faster than the connection from the clients to the data center. Hence, it would be an optimization idea for READ/GET operations to let the servers communicate with each other and partly take over the client role such that only one round of communication between a client and the data center is required.

A further optimization to the LwKVS protocol would be to employ the technique of *deduplication*. *Deduplication* [MCM01] has been used to reduce the storage and message complexity in several distributed storage implementations [BCGD00, QD02, DGH⁺09] and file systems [MCM01, DAB⁺02, KDLT04, ZLP08, CAVL09]. The idea is, before a value is written to the KVS, to first send a hash of the value to the servers requesting if the same value has already been stored. In this case, such a duplicate value does not have to be sent and stored again and a reference to the storage location of the identical value is sufficient. Applying this deduplication technique in the context of the proposed LwKVS protocol would further strike the path of *lightweight* solutions.

6 Conclusion

The thesis at hand has introduced approaches to cope with systems in untrusted environments where components may act maliciously. Two main problems have been explored. The first one deals with the question which properties can be achieved by a distributed protocol executed among clients and a server, where the server may exhibit arbitrary, malicious behavior. These properties are comprised by *fork-consistent* semantics. In this context, contributions (C1) and (C2) demonstrate how a fundamental impossibility result regarding the liveness of the implemented operations can be circumvented. The proposed protocols LINEAR and CONCUR allow operations to *abort*, enabling an easier design of systems with fork-linearizable semantics as complicated *lock-based* mechanism are no longer required. Contributions (C3) and (C4) improve on existing fork-consistent implementations by weakening fundamental assumptions on the computational power of the underlying server without sacrificing the implemented properties. Instead of having the server execute arbitrary code, the proposed protocols AFL and WFL are based on a server that implements only simple storage objects, also known as registers.

The second issue addressed by this thesis aims to reduce the influence of malicious system components by replicating the provided service over multiple servers such that the malicious behavior of a fraction of the replicas can be tolerated. In this context, contributions (C5) and (C6) improve on existing implementations of Byzantine fault-tolerant storage in vital aspects. The introduced LWR and LWKVS protocols require a minimal number of replicas, feature an optimal number of communication rounds, do not make use of self-verifying data, and support an unbounded number of possibly malicious readers which makes them very *lightweight* approaches.

The following sections briefly summarize the contributions made by this thesis and discuss open questions for future exploration.

Abortable Fork-Linearizable Storage

Implementations that provide fork-consistent semantics can be seen as gracefully-degrading systems. Although the server interacting with the clients is untrusted, it is expected to behave correctly most of the time. In this case, the clients should observe a *linearizable* history and *wait-free* operations. If the server deviates

from the specified behavior and acts maliciously, the system gracefully degrades by still providing fork-consistency. Note, that in this case the system cannot be expected to be responsive as the malicious server may simply cease to process client requests. Unfortunately, it has been shown that such behavior of graceful degradation is not possible in the asynchronous system model for implementations with *fork-linearizability* which is the strongest known fork-consistent property. A recent result of Cachin *et al.* [CSS07] states that there is no fork-linearizable storage implementation where operations are *wait-free* if the server behaves correctly. This impossibility result also explains why existing solutions providing fork-linearizability are based on locking techniques and ensure best-case liveness conditions even weaker than *obstruction-freedom*.

The idea of the proposed contributions (C1) and (C2) is to improve on liveness in case the server behaves correctly by avoiding the use of locks. Instead, the introduced LINEAR and CONCUR protocols allow the implemented operations to *abort* under concurrency. Aborting operations helps to get along with the findings of Cachin *et al.* [CSS07], as letting concurrent operations that access the same logical data complete independently can be exploited by a Byzantine server to violate fork-linearizability. The liveness condition that can be achieved with abortable operations has been shown to be at least as strong as *obstruction-freedom* [AFH⁺07]. Differently from lock-based implementations where the crash of a single client (holding the lock) may cause the whole system to block forever, in the LINEAR and CONCUR protocol a crashed client may only force a single operation to abort. From a practical point of view, implementations with abortable operations are also very attractive as it has been shown that in practical systems, that are occasionally synchronous, abortable operations can be boosted to be wait-free [AT08].

The two protocols LINEAR and CONCUR, introduced as contributions (C1) and (C2) of the thesis at hand, constitute the first lock-free emulations of fork-linearizable shared memory on a Byzantine server. Both protocols never violate fork-linearizability, the strongest existing fork-consistent property, even if the server behaves in a malicious way. If the server acts as specified, the implemented READ and WRITE operations satisfy linearizability, the strongest consistency property, and *obstruction-freedom* which means that every operation of a correct client that executes in isolation completes successfully. The LINEAR protocol is based on timestamp vectors and it has a communication complexity of $\mathcal{O}(n)$. It allows to abort operations under concurrency. The CONCUR protocol improves on the LINEAR protocol in the way how concurrent operations are handled. In the CONCUR protocol only concurrent operations accessing the *same* register of the shared memory need to be aborted. To achieve this, the CONCUR protocol relies on timestamp matrices and has a communication complexity of $\mathcal{O}(n^2)$.

Both the READ and WRITE operations implemented by the LINEAR and CONCUR

protocol need two rounds of communication between the clients and the server to complete. While the two communication rounds for `WRITE` operations are arguably necessary, the thesis at hand leaves for future work the conjecture that `READ` operations can be optimized in the the `LINEAR` and `CONCUR` protocol to complete after a single round. This would also imply that `READ` operations can be made *wait-free*.

Fork-Consistent Emulations from Registers

The protocols `LINEAR` and `CONCUR` of contributions (C1) and (C2) required the server to execute non-trivial computation steps. These comprise so-called *read-modify-write* operations where the server is able in one atomic step to update a variable with a newer value based on the current state of the variable. Such *conditional writes* [CJS12] are very helpful to implement a timestamping mechanism where the variable holding the current timestamp is only updated with higher timestamp values. In a seminal work of Herlihy [Her91], servers with such computational capabilities have been classified as being the strongest possible shared objects. They can be used to implement any shared functionality in a wait-free manner, and are *universal* in this sense. On the other end of Herlihy’s classification lie shared memory objects, called *registers*. They can be accessed only via a simple read/write interface and are computationally weaker than the *universal* objects. Even for practical systems this can make a big difference in cost because full-fledged servers or virtual machines (constituting the *universal* servers) are typically more expensive than simple disks or cloud-based key-value stores (corresponding to servers providing only *registers*).

The protocols `AFL` and `WFL` proposed as contributions (C3) and (C4) of this thesis show the surprising result that in the context of fork-consistency, the assumptions on the *universality* of the server can be abandoned in many cases and weaker read/write *registers* may be used instead without sacrificing any of the provided properties. Moreover, the `AFL` and `WFL` protocols constitute the first known fork-consistent emulations on a Byzantine server that provides only registers.

The `AFL` protocol never violates fork-linearizability and implements a shared functionality of universal type. Similar to non-fork-consistent universal implementations only from registers, the `AFL` protocol may abort operations under concurrency [AFH⁺07]. Hence, fork-linearizability may be “added” to such protocols without making additional assumptions.

The `WFL` protocol implements a shared memory that gracefully degrades to *weak* fork-linearizability in case the server providing the registers is malicious. Else, the operations implemented by the `WFL` protocol are wait-free and appear in linearizable order. Weak fork-linearizability is the strongest known fork-consistency property that allows wait-free operations in the best-case. Although

being weaker than fork-linearizability, it has shown to be of practical relevance [CKS11]. Moreover, the WFL protocol shows for the first time that registers are sufficient to implement a fork-consistent shared memory.

The thesis at hand leaves in this context two open problems for future research: The first raises the question if the AFL protocol implementing a universal type could provide a stronger liveness condition than obstruction-freedom in case the server is correct. The first conjecture is, that achieving liveness conditions stronger than obstruction-freedom with a universal construction only from registers is independent of whether fork-consistent semantics are provided or not. Secondly, the conjecture is that as in the case of the WFL protocol and by using a computationally stronger server, *wait-freedom* can be achieved for the implementation of a universal type only if *weak fork-linearizability* instead of fork-linearizability is provided.

The second open challenge would be to reduce the communication complexity of the WFL protocol. The WFL protocol uses matrices of size $n \times n$ where the size of each entry depends on the total number of operations N , and thus, has a communication complexity of $O(N \cdot n^2)$. A supposed approach to reduce this communication complexity would be to implement a “garbage collection” mechanism that avoids a dependence of the communication complexity on the total number of operations N .

Lightweight Atomic Storage

An *atomic register* is an important building block for distributed systems, as demonstrated by the AFL and WFL protocols of contributions (C3) and (C4) that are built upon this abstraction. The LWR protocol given as contribution (C5) of this thesis aims at implementing an atomic register in a Byzantine fault-tolerant manner. The basic idea is to replicate the implementation over a collection of servers such that the malicious behavior of a fraction of the replicas can be masked. Such *masking* fault-tolerance [AS85] means that even in the presence of faulty replicas all specified safety and liveness properties are satisfied. To limit the costs incurred by replication, the challenge is to achieve *optimal resilience* which is $3t+1$ replicas to tolerate t faulty ones in the context of Byzantine fault-tolerance. Specifically, fault-tolerant implementations that feature optimal resilience and provide wait-free operations are called *robust*.

In the context of robust atomic register implementations it is a challenge to achieve an optimal latency of the implemented operations. The latency is measured in the number of communication rounds that has to be performed between a client and the servers. The latency is an important aspect as it directly influences the performance of an atomic register implementation. Moreover, as the use of self-verifying data [RSA78] is considered as a significant source of overhead

[Rei94, MR97], implementations in the non-authenticated data model are desirable. However, with respect to the latency of robust, atomic register implementations there exists a fundamental difference between the authenticated and the non-authenticated data model. If the system model allows the use of self-verifying data, a robust, Byzantine fault-tolerant atomic register can be implemented where WRITE operations require a *single* and READ operations require *two* communication rounds [MR98]. In the non-authenticated data model, the fastest atomic register implementations perform *two* communication rounds during WRITE and *four*¹ during READ operations [DGM⁺11].

The LwR protocol, introduced as contribution (C5) of the thesis at hand, constitutes a robust, Byzantine fault-tolerant atomic register implementation in the non-authenticated data model where both READ and WRITE operations complete after only *two* communication rounds, thus, significantly reducing the gap to implementations in the authenticated data model. The latency of READ operations, which are usually expected to occur more frequently, achieve the same latency as in the authenticated data model. Moreover, it constitutes the first atomic register implementation not relying on self-verifying data that tolerates an unbounded number of possibly malicious readers. The reduction in latency is achieved by employing a *commitment scheme*, given here in two variants: One variant is based on a collision-resistant hash function while the other one is built upon a secret sharing scheme [Sha79], which makes the latter additionally *information theoretically secure* [LCAA07, AACL07]. As the LwR protocol features optimal resilience, an optimal communication latency, does not rely on data authentication, and additionally allows for an unbounded number of malicious readers, it is regarded as a *lightweight* approach.

The LwKVS protocol, being contribution (C6) of this thesis, is based on the LwR protocol and constitutes a *lightweight*, Byzantine fault-tolerant implementation of a key-value store (KVS). A KVS turned out to be one of the most favored storage abstraction for recent cloud services [DHJ⁺07, MTJ⁺08, ALM⁺10, LM10, CWO⁺11]. As several clients are allowed to store data under the same key, an additional round of communication with the servers is required for PUT and DELETE operations in comparison to the implementation of the WRITE operations in the LwR protocol. The GET and LIST operations implemented by the LwKVS protocol achieve a two round latency just like READ operations in the LwR protocol.

The LIST operation implemented by the LwKVS protocol achieves unlike the other operations only regular semantics in order to perform in as few as two communication rounds. This thesis leaves for future exploration the conjecture that atomic LIST operations can be implemented at the cost of sacrificing *wait-freedom* for *obstruction-freedom* (cf. the discussion in Section 5.6). A second

¹If an unbounded number of readers is supported; else *three* communication rounds.

optimization of the LwKVS protocol that goes beyond the scope of this thesis is to optimize the communication pattern of the protocol with respect to practical settings. In a real-world deployment, the servers are usually located within the same data center while clients access via the Internet. Hence, communication within the data center is much more powerful than the connection from the clients to the data center. An idea for optimization would be to let the servers communicate with each other and partly take over the client role such that only one round of communication between a client and the data center is required. As a final, open optimization to the LwKVS protocol, the technique of *deduplication* is proposed here to strike the path of *lightweight* solutions. A number of storage systems have been introduced during the past decade that reduce the storage and message complexity by removing duplicate data [BCGD00, MCM01, DAB⁺02, QD02, KDLT04, ZLP08, DGH⁺09, CAVL09]. The basic principle is that operation $\text{PUT}(key, v)$ sends value v only to the KVS, if v is and has never been stored under any key at the KVS. Deduplication can be implemented by storing value v together with its hash value $H(v)$ ². Then, during the first round of a $\text{PUT}(key, v)$ operation the client requests at each replica, by sending only $H(v)$, whether value v has already been stored — in this case v does not need to be sent again saving a significant amount of bandwidth and storage capacity.

The Final Word

The thesis at hand described problems and their solutions in the research context of untrusted system environments. Systems are referred to as untrusted if some system components may exhibit arbitrary, malicious behavior. On the highest level of abstraction the thesis explores the best possible properties that can be achieved by a distributed protocol implemented among clients and an untrusted server. These properties are comprised by the notion of *fork-consistency*. The protocols introduced as contributions (C1) – (C4) throughout this thesis fundamentally improve over existing fork-consistent implementations. The LINEAR and CONCUR protocol (contribution (C1) and (C2)) are this first fork-linearizable implementations that are not based on locks. The AFL and WFL protocols from contribution (C3) and (C4) constitute the first fork-consistent implementations that require the server only to implement simple storage objects and thereby fundamentally reduce the underlying system assumptions without sacrificing the provided properties.

On a lower level of abstraction, this thesis introduces solutions to make system components more trustful. Contributions (C5) and (C6) implement shared storage abstractions in a Byzantine fault-tolerant manner. Both, the LWR and the LwKVS protocol are *lightweight* implementations featuring optimal resilience

²Where H is a collision resistant hash function.

and latency in the non-authenticated data model by employing a commitment scheme. The atomic register and the key-value-store implemented by the LwR and the LwKVS protocol, respectively, further support an unbounded number of malicious clients which is an outstanding feature for the non-authenticated data model.

Bibliography

- [AAB07] Amitanand S. Aiyer, Lorenzo Alvisi, and Rida A. Bazzi. Bounded Wait-free Implementation of Optimally Resilient Byzantine Storage Without (Unproven) Cryptographic Assumptions. In *Proceedings of the 21st International Symposium on Distributed Computing (DISC)*, Lemesos, Cyprus, 2007.
- [AACL07] Amitanand S. Aiyer, Lorenzo Alvisi, Allen Clement, and Harry C. Li. Information-Theoretically Secure Byzantine Paxos. Technical Report TR-07-21, The University of Texas at Austin, May 2007. Available online at <http://www.cs.utexas.edu/users/anand/pubs/aiyer07Information.pdf>.
- [ABND95] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing Memory Robustly in Message-Passing Systems. *J. ACM*, 42(1):124–142, January 1995.
- [ACKM06] Ittai Abraham, Gregory Chockler, Idit Keidar, and Dahlia Malkhi. Byzantine Disk Paxos: Optimal Resilience with Byzantine Shared Memory. *Distributed Computing*, 18(5):387–408, 2006.
- [ACKM07] Ittai Abraham, Gregory Chockler, Idit Keidar, and Dahlia Malkhi. Wait-free Regular Storage from Byzantine Components. *Inf. Process. Lett.*, 101(2):60–65, January 2007.
- [AFH⁺07] Marcos K. Aguilera, Svend Frolund, Vassos Hadzilacos, Stephanie L. Horn, and Sam Toueg. Abortable and Query-Abortable Objects and Their Efficient Implementation. In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 23–32, Portland, Oregon, USA, 2007. ACM.
- [AGHK09] Hagit Attiya, Rachid Guerraoui, Danny Hendler, and Petr Kuznetsov. The Complexity of Obstruction-Free Implementations. *J. ACM*, 56(4):1–33, 2009.
- [AGK05] Hagit Attiya, Rachid Guerraoui, and Petr Kouznetsov. Computing with Reads and Writes in the Absence of Step Contention. In *Proceed-*

- ings of the 19th International Symposium on Distributed Computing (DISC)*, pages 122–136, Cracow, Poland, 2005.
- [AGR08] Hagit Attiya, Rachid Guerraoui, and Eric Ruppert. Partial Snapshot Objects. In *Proceedings of 20th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 336–343, Munich, Germany, 2008.
 - [AKMS11] Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. Dynamic Atomic Storage Without Consensus. *J. ACM*, 58(2):7:1–7:32, April 2011.
 - [ALM⁺10] Eric Anderson, Xiaozhou Li, Arif Merchant, Mehul A. Shah, Kevin Smathers, Joseph Tucek, Mustafa Uysal, and Jay J. Wylie. Efficient Eventual Consistency in Pahoehoe, an Erasure-Coded Key-blob Archive. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 181–190, Chicago, Illinois, USA, 2010.
 - [AS85] Bowen Alpern and Fred B. Schneider. Defining Liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985.
 - [AT08] Marcos K. Aguilera and Sam Toueg. Timeliness-Based Wait-Freedom: A Gracefully Degrading Progress Condition. In *Proceedings of the 27th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 305–314, Toronto, Canada, 2008. ACM.
 - [AWS] AWS. Amazon Web Services. Website. Available online at <http://aws.amazon.com/>; visited September 2012.
 - [BCC88] Gilles Brassard, David Chaum, and Claude Crépeau. Minimum Disclosure Proofs of Knowledge. *J. Comput. Syst. Sci.*, 37(2):156–189, October 1988.
 - [BCE⁺11] Cristina Basescu, Christian Cachin, Ittay Eyal, Robert Haas, and Marko Vukolic. Robust Data Sharing with Key-Value Stores. In *Proceedings of the 30th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 221–222, San Jose, California, USA, 2011. ACM.
 - [BCE⁺12] Cristina Basescu, Christian Cachin, Ittay Eyal, Robert Haas, Alessandro Sorniotti, Marko Vukolic, and Ido Zachevsky. Robust Data Sharing with Key-Value Stores. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, Boston, MA, USA, June 2012.

-
- [BCGD00] William J. Bolosky, Scott Corbin, David Goebel, and John R. Douceur. Single Instance Storage in Windows 2000. In *Proceedings of the 4th conference on USENIX Windows Systems Symposium - Volume 4*, WSS'00, Seattle, Washington, 2000. USENIX Association.
- [BD04] Rida A. Bazzi and Yin Ding. Non-skipping Timestamps for Byzantine Data Storage Systems. In *Proceedings of the 18th International Symposium on Distributed Computing (DISC)*, pages 405–419, Amsterdam, Netherlands, 2004.
- [Cac11] Christian Cachin. Integrity and Consistency for Untrusted Services. In *Proceedings of the 37th International Conference on Current Trends in Theory and Practice of Computer Science*, SOFSEM'11, pages 1–14, Nový Smokovec, Slovakia, 2011. Springer-Verlag.
- [CAVL09] Austin T. Clements, Irfan Ahmad, Murali Vilayannur, and Jinyuan Li. Decentralized Deduplication in SAN Cluster File Systems. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, San Diego, California, 2009. USENIX Association.
- [CG09] Christian Cachin and Martin Geisler. Integrity Protection for Revision Control. In *Proceedings of the 7th International Conference on Applied Cryptography and Network Security*, ACNS '09, pages 382–399, Paris-Rocquencourt, France, 2009. Springer-Verlag.
- [CGK07] Gregory Chockler, Rachid Guerraoui, and Idit Keidar. Amnesic Distributed Storage. In *Proceedings of the 21st International Symposium on Distributed Computing (DISC)*, pages 139–151, Lemesos, Cyprus, 2007.
- [CJS12] Christian Cachin, Birgit Junker, and Alessandro Sorniotti. On Limitations of Using Cloud Storage for Data Replication. In *Proceedings of the 6th Workshop on Recent Advances in Intrusion Tolerance and reSilience*, WRAITS'12, Boston, MA, USA, 2012.
- [CKS09a] Christian Cachin, Idit Keidar, and Alexander Shraer. Fork Sequential Consistency is Blocking. *Inf. Process. Lett.*, 109(7):360–364, 2009.
- [CKS09b] Christian Cachin, Idit Keidar, and Alexander Shraer. Trusting the Cloud. *ACM SIGACT News, Distributed Computing in the Clouds*, 40(2):81–86, June 2009.
- [CKS11] Christian Cachin, Idit Keidar, and Alexander Shraer. Fail-Aware Untrusted Storage. *SIAM Journal on Computing*, 40(2):493–533, April 2011.

- [CSS07] Christian Cachin, Abhi Shelat, and Alexander Shraer. Efficient Fork-Linearizable Access to Untrusted Shared Memory. In *Proceedings of the 26th ACM Symposium on Principles of Distributed computing (PODC)*, pages 129–138, Portland, Oregon, USA, 2007. ACM.
- [CT96] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *J. ACM*, 43(2):225–267, 1996.
- [CT06] Christian Cachin and Stefano Tessaro. Optimal Resilience for Erasure-Coded Byzantine Distributed Storage. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 115–124, Philadelphia, PA, USA, 2006. IEEE Computer Society.
- [CVS] CVS. Concurrent Versions System. Website. Available online at <http://www.nongnu.org/cvs/>; visited September 2012.
- [CWO⁺11] Brad Calder, Ju Wang, Aaron Ogus, Niranjana Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows Azure Storage: A Highly Available Cloud Storage Service With Strong Consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, SOSP '11*, pages 143–157, Cascais, Portugal, 2011. ACM.
- [DAB⁺02] John R. Douceur, Atul Adya, William J. Bolosky, Dan Simon, and Marvin Theimer. Reclaiming Space from Duplicate Files in a Serverless Distributed File System. In *The 22nd International Conference on Distributed Computing Systems (ICDCS)*, pages 617–624, Vienna, Austria, 2002.
- [DGH⁺09] Cezary Dubnicki, Leszek Gryz, Lukasz Heldt, Michal Kaczmarczyk, Wojciech Kilian, Przemyslaw Strzelczak, Jerzy Szczepkowski, Cristian Ungureanu, and Michal Welnicki. HYDRAstor: a Scalable Secondary Storage. In *Proceedings of the 7th conference on File and Storage Technologies, FAST '09*, pages 197–210, San Francisco, CA, USA, 2009. USENIX Association.
- [DGLC04] Partha Dutta, Rachid Guerraoui, Ron R. Levy, and Arindam Chakraborty. How Fast Can a Distributed Atomic Read Be? In

- Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing*, PODC '04, pages 236–245, St. John's, Newfoundland, Canada, 2004. ACM.
- [DGLV05] Partha Dutta, Rachid Guerraoui, Ron R. Levy, and Marko Vukolic. How Fast can a Distributed Atomic Read be? Technical report, EPFL, Lausanne, Switzerland, 2005. Available online at <http://infoscience.epfl.ch/record/63240/files/paper.pdf>.
- [DGLV10] Partha Dutta, Rachid Guerraoui, Ron R. Levy, and Marko Vukolić. Fast Access to Distributed Atomic Memory. *SIAM J. Comput.*, 39(8):3752–3783, December 2010.
- [DGM⁺11] Dan Dobre, Rachid Guerraoui, Matthias Majuntke, Neeraj Suri, and Marko Vukolić. The Time-Complexity of Robust Atomic Storage. In *Proceedings of the 30th ACM Symposium on Principles of Distributed Computing*, PODC '11, pages 59–68, San Jose, California, USA, 2011. ACM.
- [DGM⁺12] Dan Dobre, Rachid Guerraoui, Matthias Majuntke, Neeraj Suri, and Marko Vukolić. The Complexity of Robust Storage. *J. ACM*, 2012. (Submitted).
- [DHJ⁺07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Voshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, Stevenson, Washington, USA, 2007. ACM.
- [DMS08] Dan Dobre, Matthias Majuntke, and Neeraj Suri. On the Time-Complexity of Robust and Amnesic Storage. In *Proceedings of 12th International Conference On Principles Of Distributed Systems (OPODIS)*, pages 197–216, Luxor, Egypt, 2008.
- [DMSS09] Dan Dobre, Matthias Majuntke, Marco Serafini, and Neeraj Suri. Efficient Robust Storage Using Secret Tokens. In *Proceedings of the 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, SSS '09, pages 269–283, Lyon, France, 2009. Springer-Verlag.
- [DMSS10] Dan Dobre, Matthias Majuntke, Marco Serafini, and Neeraj Suri. HP: Hybrid Paxos for WANs. In *Proceedings of the 2010 European De-*

- pendable Computing Conference*, EDCC '10, pages 117–126, Valencia, Spain, 2010. IEEE Computer Society.
- [EGM⁺09] Burkhard Englert, Chryssis Georgiou, Peter M. Musial, Nicolas Nicolaou, and Alexander A. Shvartsman. On the Efficiency of Atomic Multi-reader, Multi-writer Distributed Memory. In *Proceedings of the 13th International Conference on Principles of Distributed Systems*, OPODIS '09, pages 240–254, Nîmes, France, 2009. Springer-Verlag.
- [Fic05] Faith Ellen Fich. How Hard Is It to Take a Snapshot? In *Proceedings of the 31st International Conference on Current Trends in Theory and Practice of Computer Science*, SOFSEM'05, pages 28–37, Liptovsky Jan, Slovak Republic, 2005.
- [FL03] Rui Fan and Nancy Lynch. Efficient Replication of Large Data Objects. In *Proceedings of the 17th International Symposium on Distributed Computing (DISC)*, pages 75–91, Sorrento, Italy, 2003.
- [FLMS05] Faith Ellen Fich, Victor Luchangco, Mark Moir, and Nir Shavit. Obstruction-Free Algorithms Can Be Practically Wait-free. In *Proceedings of the 19th International Conference on Distributed Computing (DISC)*, pages 78–92, Cracow, Poland, 2005. Springer-Verlag.
- [FZFF10] Ariel J. Feldman, William P. Zeller, Michael J. Freedman, and Edward W. Felten. SPORC: Group Collaboration on Untrusted Resources. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, BC, October 2010.
- [GLV06] Rachid Guerraoui, Ron R. Levy, and Marko Vukolic. Lucky Read/Write Access to Robust Atomic Storage. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 125–136, Philadelphia, PA, USA, 2006. IEEE Computer Society.
- [GNS09] Chryssis Georgiou, Nicolas C. Nicolaou, and Alexander A. Shvartsman. Fault-tolerant Semifast Implementations of Atomic Read/Write Registers. *J. Parallel Distrib. Comput.*, 69(1):62–79, January 2009.
- [Goo] Google Inc. Google Docs. Website. Available online at <http://docs.google.com>; visited September 2012.
- [GV06] Rachid Guerraoui and Marko Vukolić. How Fast Can a Very Robust Read Be? In *Proceedings of the 25th ACM Symposium on Principles of Distributed Computing*, PODC '06, pages 248–257, Denver, Colorado, USA, 2006. ACM.

- [GV07] Rachid Guerraoui and Marko Vukolić. Refined Quorum Systems. In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 119–128, Portland, Oregon, USA, 2007. ACM.
- [GWGR04] Garth R. Goodson, Jay J. Wylie, Gregory R. Ganger, and Michael K. Reiter. Efficient Byzantine-Tolerant Erasure-Coded Storage. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, Florence, Italy, 2004. IEEE Computer Society.
- [Her91] Maurice Herlihy. Wait-Free Synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.
- [HGR07] James Hendricks, Gregory R. Ganger, and Michael K. Reiter. Low-Overhead Byzantine Fault-Tolerant Storage. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 73–86, Stevenson, Washington, USA, 2007. ACM.
- [HLM03] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-Free Synchronization: Double-Ended Queues as an Example. In *The 23rd International Conference on Distributed Computing Systems (ICDCS)*, Providence, RI, USA, 2003. IEEE Computer Society.
- [HS11] Maurice Herlihy and Nir Shavit. On the Nature of Progress. In *Proceedings of the 15th International Conference on Principles of Distributed Systems*, OPODIS '11, pages 313–328, Toulouse, France, 2011.
- [HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
- [IS92] Amos Israeli and Amnon Shaham. Optimal Multi-Writer Multi-Reader Atomic Register. In *Proceedings of the 11th ACM Symposium on Principles of Distributed Computing*, PODC '92, pages 71–82, Vancouver, British Columbia, Canada, 1992. ACM.
- [JCT98] Prasad Jayanti, Tushar Deepak Chandra, and Sam Toueg. Fault-tolerant Wait-free Shared Objects. *J. ACM*, 45(3):451–500, 1998.
- [KDLT04] Purushottam Kulkarni, Fred Douglass, Jason LaVoie, and John M. Tracey. Redundancy Elimination Within Large Collections of Files. In *Proceedings of the USENIX Annual Technical Conference, ATEC '04*, Boston, MA, USA, 2004. USENIX Association.

- [KRS88] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. Efficient Synchronization of Multiprocessors with Shared Memory. *ACM Trans. Program. Lang. Syst.*, 10(4):579–601, October 1988.
- [Lam86] Leslie Lamport. On Interprocess Communication. *Distributed Computing*, 1(2):77–101, 1986.
- [LCAA07] Harry C. Li, Allen Clement, Amitanand S. Aiyer, and Lorenzo Alvisi. The Paxos Register. In *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems*, SRDS '07, pages 114–126, Beijing, China, 2007. IEEE Computer Society.
- [LKMS04] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. Secure Untrusted Data Repository (SUNDR). In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 121–136, San Francisco, CA, USA, 2004.
- [LM07] Jinyuan Li and David Mazières. Beyond One-Third Faulty Replicas in Byzantine Fault Tolerant Systems. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI)*, Cambridge, Massachusetts, USA, 2007.
- [LM10] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.
- [LR06] Barbara Liskov and Rodrigo Rodrigues. Tolerating Byzantine Faulty Clients in a Quorum System. In *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, ICDCS '06, Lisboa, Portugal, 2006. IEEE Computer Society.
- [LS02] Nancy A. Lynch and Alexander A. Shvartsman. RAMBO: A Reconfigurable Atomic Memory Service for Dynamic Networks. In *Proceedings of the 16th International Conference on Distributed Computing (DISC)*, pages 173–190, Toulouse, France, 2002. Springer-Verlag.
- [Lyn98] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1998.
- [MAD02] Jean-Philippe Martin, Lorenzo Alvisi, and Michael Dahlin. Minimal Byzantine Storage. In *Proceedings of the 16th International Conference on Distributed Computing*, DISC '02, pages 311–325, Toulouse, France, 2002. Springer-Verlag.

- [MCM01] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A Low-Bandwidth Network File System. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, SOSP '01, pages 174–187, Banff, Alberta, Canada, 2001. ACM.
- [MDCS11] Matthias Majuntke, Dan Dobre, Christian Cachin, and Neeraj Suri. Fork-Consistent Constructions from Registers. In *Proceedings of the 15th International Conference On Principles Of Distributed Systems (OPODIS)*, pages 283–298, Toulouse, France, 2011.
- [MDS11] Matthias Majuntke, Dan Dobre, and Neeraj Suri. Fork-Consistent Constructions from Registers. In *Proceedings of the 30th ACM Symposium on Principles of Distributed Computing*, PODC '11, pages 209–210, San Jose, California, USA, 2011. ACM.
- [MDS12] Matthias Majuntke, Dan Dobre, and Neeraj Suri. Lightweight Robust Atomic Storage. In *Technical Report. Technische Universität Darmstadt. TR-TUD-DEEDS-07-01-2012*, July 2012.
- [MDSS09] Matthias Majuntke, Dan Dobre, Marco Serafini, and Neeraj Suri. Abortable Fork-Linearizable Storage. In *Proceedings of the 13th International Conference on Principles of Distributed Systems*, OPODIS '09, pages 255–269, Nîmes, France, 2009. Springer-Verlag.
- [MG11] Peter Mell and Tim Grance. The NIST Definition of Cloud Computing. Technical report, National Institute of Standards and Technology (NIST), January 2011. File available online at http://csrc.nist.gov/publications/drafts/800-145/Draft-SP-800-145_cloud-definition.pdf.
- [MR97] Dahlia Malkhi and Michael K. Reiter. A High-Throughput Secure Reliable Multicast Protocol. *J. Comput. Secur.*, 5(2):113–127, March 1997.
- [MR98] Dahlia Malkhi and Michael K. Reiter. Byzantine Quorum Systems. *Distributed Computing*, 11(4):203–213, 1998.
- [MR00] Dahlia Malkhi and Michael K. Reiter. An Architecture for Survivable Coordination in Large Distributed Systems. *IEEE Trans. on Knowl. and Data Eng.*, 12(2):187–202, March 2000.
- [MS02] David Mazières and Dennis Shasha. Building Secure File Systems out of Byzantine Storage. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing (PODC)*, pages 108–117, Monterey, California, USA, 2002. ACM.

- [MTJ⁺08] John MacCormick, Chandramohan A. Thekkath, Marcus Jager, Kristof Roomp, Lidong Zhou, and Ryan Peterson. Niobe: A Practical Replication Protocol. *Trans. Storage*, 3(4):1:1–1:43, February 2008.
- [OR06] Alina Oprea and Michael K. Reiter. On Consistency of Encrypted Files. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, pages 254–268, Stockholm, Sweden, 2006.
- [PSL80] M. Pease, R. Shostak, and L. Lamport. Reaching Agreement in the Presence of Faults. *J. ACM*, 27(2):228–234, 1980.
- [QD02] Sean Quinlan and Sean Dorward. Venti: A New Approach to Archival Storage. In *Proceedings of the 1st USENIX conference on File and Storage Technologies*, FAST’02, Monterey, CA, USA, 2002. USENIX Association.
- [Rei94] Michael K. Reiter. Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, CCS ’94, pages 68–80, Fairfax, Virginia, USA, 1994. ACM.
- [RS04] Phillip Rogaway and Thomas Shrimpton. Cryptographic Hash-Function Basics: Definitions, Implications and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance. *IACR Cryptology ePrint Archive*, 2004:35, 2004.
- [RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.
- [SBD⁺10] Marco Serafini, Peter Bokor, Dan Dobre, Matthias Majuntke, and Neeraj Suri. Scrooge: Reducing the Costs of Fast Byzantine Replication in Presence of Unresponsive Replicas. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, Chicago, Illinois, USA, 2010.
- [SCC⁺10] Alexander Shraer, Christian Cachin, Asaf Cidon, Idit Keidar, Yan Michalevsky, and Dani Shaket. Venus: Verification for Untrusted Cloud Storage. In *Proceedings of the 2010 ACM Workshop on Cloud Computing Security*, CCSW ’10, pages 19–30, Chicago, Illinois, USA, 2010. ACM.

- [SDM⁺10] Marco Serafini, Dan Dobre, Matthias Majuntke, Péter Bokor, and Neeraj Suri. Eventually Linearizable Shared Objects. In *Proceedings of the 29th ACM Symposium on Principles of Distributed Computing*, PODC '10, pages 95–104, Zurich, Switzerland, 2010. ACM.
- [SFV⁺04] Yasushi Saito, Svend Frølund, Alistair Veitch, Arif Merchant, and Susan Spence. FAB: Building Distributed Enterprise Disk Arrays From Commodity Components. *SIGARCH Comput. Archit. News*, 32(5):48–58, October 2004.
- [SH02] Frank Schmuck and Roger Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, Monterey, CA, USA, 2002. USENIX Association.
- [Sha79] Adi Shamir. How to Share a Secret. *Commun. ACM*, 22(11):612–613, November 1979.
- [SVN] SVN. Subversion. Website. Available online at <http://subversion.apache.org/>; visited September 2012.
- [Tau09] Gadi Taubenfeld. On the Computational Power of Shared Objects. In *Proceedings of the 13th International Conference on Principles of Distributed Systems*, OPODIS '09, pages 270–284, Nîmes, France, 2009. Springer-Verlag.
- [Val94] John D. Valois. Implementing Lock-Free Queues. In *Proceedings of the 7th International Conference on Parallel and Distributed Computing Systems (PDCS)*, pages 64–69, Las Vegas, NV, USA, 1994.
- [Whi97] E. James Whitehead, Jr. World Wide Web Distributed Authoring and Versioning (WebDAV): An Introduction. *StandardView*, 5(1):3–8, 1997.
- [Wik] Wikipedia. List of file systems, distributed file systems. Website. Available online at http://en.wikipedia.org/wiki/List_of_file_systems; visited September 2012.
- [WSS09] Peter Williams, Radu Sion, and Dennis Shasha. The Blind Stone Tablet: Outsourcing Durability to Untrusted Parties. In *Proceedings of the 16th Annual Network & Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, 2009.

- [YWG⁺08] Jiangming Yang, Haixun Wang, Ning Gu, Yiming Liu, Chunsong Wang, and Qiwei Zhang. Lock-free Consistency Control for Web 2.0 Applications. In *Proceedings of the 17th International World Wide Web Conference (WWW)*, pages 725–734, Beijing, China, 2008. ACM.
- [ZLP08] Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies, FAST'08*, pages 18:1–18:14, San Jose, California, USA, 2008. USENIX Association.

List of Algorithms

3.1	LINEAR/CONCUR Protocol: Read/Write Operations of the Clients .	41
3.2	LINEAR Protocol, Algorithm of the Server	41
3.3	LINEAR Protocol, Algorithm of the Clients	43
3.4	CONCUR Protocol, Algorithm of the Server	52
3.5	CONCUR Protocol, Algorithm of the Clients	53
4.1	AFL Protocol, Algorithm of the Clients	73
4.2	INC&READ Counter, Algorithm of the Clients	75
4.3	WFL Protocol, Variables of the Clients	83
4.4	WFL Protocol, Algorithm of the Clients	84
5.1	LwR Protocol, Algorithm of the Writer	107
5.2	LwR/LwKVS Protocol, Writer Procedures with Secret Sharing . .	108
5.3	LwR/LwKVS Protocol, Writer Procedures with Hash Function . .	109
5.4	LwR Protocol, Algorithm of the Readers	110
5.5	LwR Protocol, Algorithm of the Servers	112
5.6	LwR Protocol, VERIFY Procedure with Hash Function	112
5.7	LwR Protocol, VERIFY Procedure with Secret Sharing	112
5.8	LwKVS Protocol, PUT and DELETE Operations of the Clients . . .	122
5.9	LwKVS Protocol, GET Operation of the Clients	124
5.10	LwKVS Protocol, LIST Operation of the Clients	125
5.11	LwKVS Protocol, Algorithm of the Servers	126
5.12	LwKVS Protocol, Algorithm of the Servers (cont.)	127
5.13	LwKVS Protocol, VERIFY Procedure with Hash Function	127
5.14	LwKVS Protocol, VERIFY Procedure with Secret Sharing	128

List of Figures

1.1	Intuition behind Cachin's impossibility result	4
1.2	Intuition behind <i>abortable</i> operations	5
1.3	Overview on Contributions (C1) – (C4)	11
1.4	Basic principle implemented by the WFL protocol	21
3.1	LINEAR/CONCUR: Two indistinguishable executions	61
4.1	Correctness Idea of the AFL Protocol	74
4.2	Proof of Lemma 4.13	80
4.3	Basic principle implemented by the WFL protocol	82
4.4	Correctness Idea of the WFL Protocol	86
4.5	Proof of Corollary 4.20	90
5.1	Impossibility of writing in one round	104
5.2	Output of operation LIST() violating atomicity	119

List of Figures

Curriculum Vitae

Matthias Majuntke was born on November 16th, 1979 in Dernbach (Westerwald), Germany. He received the Abitur from Konrad-Adenauer-Gymnasium in Westerburg, Germany, in June 1999. Matthias performed the alternative civilian service from September 1999 till July 2000. In October 2000, he started his computer science studies at Technische Universität Clausthal, Germany, which he completed with the degree of Vordiplom in Summer 2002. Matthias continued his computer science studies at RWTH Aachen University, Germany, in October 2002. He graduated and obtained his Dipl.-Inform. degree in March 2006. After his graduation, in September 2006 Matthias joined the DEEDS group at the computer science department of Technische Universität Darmstadt, Germany, and started his doctoral studies under the supervision of Prof. Neeraj Suri, PhD. Matthias was member of the international Research Training Group on “Cooperative, Adaptive and Responsive Monitoring in Mixed Mode Environments” and his research has been funded by a scholarship of the German Research Foundation (DFG) from September 2006 to January 2009. During September 2006 to July 2012, besides his work as a researcher, Matthias has served as teaching assistant and as reviewer for several international conferences and journals.